GUILHERME MORAIS LOPES DOS SANTOS

INVESTIGAÇÃO SOBRE A DIFUSÃO DE PROCESSOS E PRÁTICAS DE ENGENHARIA, ARQUITETURA E DESIGN DE SOFTWARE E SEUS IMPACTOS EM EMPRESAS DE TECNOLOGIA

(*versão pré-defesa, compilada em 11 de maio de 2022*)

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Andrey Pimentel.

CURITIBA PR

2022

RESUMO

A engenharia de software surge na década de 1960 como tentativa de desenvolver uma abordagem sistemática e de qualidade mensurável para projetos de software da época, que sofriam com dificuldades em estimativas de prazo, orçamento, e com uma base de código difícil de manter e evoluir. Este trabalho dedica-se a explorar as práticas e processos desenvolvidos a partir da chamada Crise do Software dos anos 1970, bem como a difusão dos mesmos em empresas de tecnologia nos dias atuais. Discute, ainda, os resultados da pesquisa feita com 33 profissionais da área, bem como os efeitos práticos da aplicação da engenharia, arquitetura e design de software em projetos reais.

Palavras-chave: Engenharia de Software. Design de Software. Arquitetura de Software. Ciência da Computação. Empresas de Tecnologia. Boas práticas de desenvolvimento de software.

ABSTRACT

Software engineering emerged in the 1960s as an attempt to develop a more systematic approach with measurable quality to software projects of the time, which suffered from difficulties in estimating deadlines, budget, and with a code base difficult to maintain and evolve. This work is dedicated to exploring the practices and processes developed since the so-called Crisis of Software of the 1970s, as well as their diffusion in technology companies today. It also explores the results of the research made with 33 professionals of the area, as well as the practical effects of applying software engineering, architecture and design in real projects.

Keywords: Software Engineering. Software Design. Software Architecture. Computer Science. Technology Companies. Software development good practices.

LISTA DE FIGURAS

LISTA DE ACRÔNIMOS

TDD	Test Driven Development
CI/CD	Continuos Integration and Continuous Delivery
SAAS	Software As A Service
SOLID	Single responsibility principle, Open-closed principle, Liskov substitution principle, Interface segregation principle, Dependency inversion principle
IDE	Integrated Development Environment
PR	Pull Request ou Merge Request
API	Application Programming Interface

SUMÁRIO

1 INTRODUÇÃO

Nos anos 1970, o expressivo crescimento da demanda por softwares — e a imaturidade da engenharia de software — deu origem à chamada "Crise do Software", termo utilizado para se referir à crescente complexidade dos desafios a serem resolvidos utilizando programação e tecnologia, bem como aos diversos problemas dos projetos na época. O termo foi cunhado por Edsger Dijkstra em 1972 na apresentação *The Humble Programmer*, publicada no periódico *Communications of the ACM* (Dijkstra, 1972). A Crise do Software é caracterizada pela a dificuldade das equipes de engenheiros em estimar o custo e o tempo de desenvolvimento dos projetos de forma assertiva, bem como a não satisfação dos requisitos levantados. A dificuldade em manter e evoluir o código era, no entanto, a principal característica da crise. Este trabalho dedica-se a explorar os principais processos e práticas da engenharia, arquitetura e design de software, que foram desenvolvidos a partir da Crise do Software, a fim de tornar estas disciplinas mais sistemáticas e desenvolver maneiras objetivas de medir a qualidade do software; bem como discutir a difusão destas práticas nos dias de hoje dentro das empresas de tecnologia no Brasil.

Os resultados da pesquisa desta monografia (abordados na sessão Resultados e discussão), aplicada à 33 profissionais de grandes empresas de tecnologia no Brasil, apontam que 27,3% dos desenvolvedores dizem não possuir tempo o suficiente para aplicar processos, seguidos de 15,1% que dizem não enxergar a necessidade de práticas listadas, e 15,1% que afirmam não conhecê-las. Este artigo explora, ainda, as consequências práticas destes resultados, bem como os impactos objetivos da engenharia, arquitetura e do design de software quando seus processos e práticas são observados e aplicados em projetos reais.

Na sessão Conceitos Básicos, são listadas e explicadas cada uma das práticas e processos analisados, explorando a motivação, objetivos, impactos, e exemplos de aplicação de cada um. A sessão Metodologia discorre sobre a metodologia utilizada na pesquisa.

1.1 CONTEXTO E PROBLEMA

A Crise do Software dos anos 1970, doravante referida como “Crise", deu origem ao esforço de criar abordagens sistemáticas e de qualidade mensurável para as sub-áreas da Ciência da Computação voltadas à engenharia, design, arquitetura, qualidade e teste de software. A Crise era principalmente caracterizada pelas seguintes situações:

- Projetos que ultrapassavam o orçamento previsto;
- projetos que ultrapassavam a data de entrega prevista;
- ineficiência do software desenvolvido;
- baixa qualidade do software desenvolvido;

- não satisfação dos requisitos levantados;
- dificuldade de manutenção no código dos projetos; e
- projetos cancelados antes da entrega.

De forma quase unânime, projetos que passam por situações como as listadas acima não possuem métricas efetivas e objetivas de qualidade (Jones, 2004). Ainda que algumas das situações tenham outras causas (como, por exemplo, os projetos cancelados antes da entrega, que frequentemente tratam-se de casos de "*Vaporware*"— termo cunhado em 1982 para se referir a projetos anunciados ao público, porém nunca entregues, a fim de ajudar a evitar que os usuários migrem para softwares concorrentes), a falta de métricas de qualidade está intrinsecamente ligada com à não observação das práticas e processos desenvolvidos após a Crise, causa direta dos problemas listados. Entre as práticas mais comumente ignoradas, seis ganham destaque: falta de planejamento, falta de assertividade na estimativa de custo, falta de métricas, não instituição de metas e objetivos concretos ao longo do projeto, dificuldade de controlar mudanças ao longo do desenvolvimento, e negligência quanto ao controle de qualidade. Estas práticas são conceitos amplos que podem ser explorados um a um a fim de obter uma maior granularidade de processos e práticas objetivas e de qualidade mensurável que podem ser aplicados nos projetos de software a fim de evitar os problemas que surgiram durante a Crise, mas que persistem até hoje.

A sessão Conceitos Básicos lista e descreve práticas derivadas do planejamento, da instituição de métricas, e principalmente do controle de qualidade, para em seguida explorar como a observação e implementação destes processos e práticas ajudam a mitigar os problemas citados.

1.2 OBJETIVO

A pesquisa feita com 33 profissionais da área de tecnologia da informação tem como objetivo ajudar a elucidar o estado atual das práticas relacionadas a controle de qualidade, métricas e planejamento de projeto nas empresas de tecnologia brasileiras, para então defender a ligação de causa e consequência entre a frequente não observação e instituição dos processos e práticas descritas e a baixa qualidade do software entregue, evidenciada pelos problemas causadores da Crise e que ainda persistem nos dias de hoje em diversas companhias.

Como objetivo secundário, ainda, propõe a difusão das práticas ao listar, explicar e discutir cada uma delas, já que 48,4% dos participantes da pesquisa não conhecem a maioria dos conceitos listados. Entretanto, apenas conhecer as práticas não parece o suficiente para que elas sejam de fato aplicadas, já que 72,8% dos participantes concordam que gostariam de aplicar mas não possuem tempo o suficiente. Portanto, discute-se, ainda — a fim de desmistificar —, a difundida afirmação de que a observação e instituição de práticas e processos de engenharia, design e arquitetura de software aumentam o tempo total de desenvolvimento do projeto.

2 CONCEITOS BÁSICOS

Antes de discutir os resultados da pesquisa para estabelecer a ligação de causa e consequência entre a não observação de práticas relacionadas a planejamento, métricas, controle de qualidade e os problemas causadores da Crise, é necessário listar e descrever cada uma das práticas abordadas no questionário. Esta sessão está relacionada, ainda, ao objetivo de difundir os conceitos, mas sem pretensão de se aprofundar em cada um deles, visto que há materiais — inclusive originais —, amplamente difundidos que cumprem melhor este trabalho.

2.1 CÓDIGO LIMPO (*CLEAN CODE*)

Introduzido inicialmente por Robert C. Martin em seu livro *Clean Code* (Martin, 2008), trate-se do ato de escrever código legível, reusável e fácil de dar manutenção. O conceito foi amplamente abraçado pela comunidade de engenheiros de software ao redor do mundo, também evidenciado pelo resultado da pesquisa deste trabalho, onde 81,8% dos participantes afirmam conhecer e aplicar as práticas sugeridas pelo Código Limpo. A comunidade, ainda, produz material à respeito estendendo as práticas recomendadas pelos autores do livro original. Os princípios do Código Limpo não tem pretensão de serem estritamente aceitos e aplicados, já que não há consenso universal sobre eles. Restringem-se apenas ao papel de diretrizes de codificação, resultado dos anos de experiencia dos autores do livro, já que a engenharia de software tem apenas em torno de 50 anos e ainda está em desenvolvimento.

Entender e aplicar Código Limpo em projetos de software não faz com que o engenheiro seja melhor, e usá-lo durante anos não significa que o código estará livre de problemas. Neste contexto, uma comum analogia é de que o software deve ser tratado como "argila molhada", que começa como um rascunho que necessita ser moldado a fim de chegar à sua forma final. As práticas de Código Limpo aqui citadas são resultado da leitura de guias da comunidade específicos para diferentes linguagens de programação, e os exemplos de código estão escritos em Go. Entretanto, todas as práticas podem ser replicadas para quase qualquer linguagem, com exceção, talvez, às linguagens de baixo nível como Assembly — que raramente são utilizadas em empresas de tecnologia, como podemos observar nos resultados do *Developer Survey* 2021 do *Stack Overflow* (Overflow, 2021), que, entre outros temas, estabelece um ranking de popularidade das linguagens entre os oitenta mil entrevistados.

2.1.1 Variáveis

Nomes de variáveis são frequentemente negligenciados pelos engenheiros, dificultando o entendimento do código. Em uma análise de 10 projetos *open-source* aleatoriamente escolhidos no *GitHub*, foi comum encontrar variáveis cujo nome está abreviado, impronunciável, ou até

com apenas um caractere. Também é comum observar nomes que não permitem inferir o que de fato é o conteúdo da variável, obrigando o leitor a fazer "mapeamento mental", dificultando a leitura e entendimento. O Código Limpo sugere diretrizes a serem seguidas ao nomear variáveis.

2.1.1.1 Nomes significativos e pronunciáveis

Nomes de variáveis devem explicar o conteúdo da variável dentro do contexto/domínio do programa. Devem, ainda, ser pronunciáveis, facilitando o entendimento do código (ao evitar "mapeamento mental" do significado e do conteúdo das variáveis), a busca por palavras-chave no código fonte, e a discussão sobre os termos dentro da equipe de desenvolvimento. Abaixo, um exemplo de uma *struct* em Go que representa um sistema linear, cujos nomes não são significativos:

```
1 type LinSist_t struct {
2     N uint      // Linear System dimension
3     M *[][]float64 // Matrix
4     B *[]float64   // Independent Terms
5 }
```

E, abaixo, um exemplo de como seria a mesma *struct* se os nomes fossem significativos:

```
1 type LinearSystem struct {
2     Dimension        uint
3     Matrix           *[][]float64
4     IndependentTerms *[]float64
5 }
```

Observe, ainda, que utilizar nomes significativos também torna desnecessário o uso de comentários no código. O Código Limpo defende que comentários não são necessários quando o código "fala por si mesmo". O uso de comentários não é, entretanto, "proibido" pelas diretrizes, apenas restrito à locais onde são de fato necessários.

2.1.1.2 Relação entre o vocabulário utilizado e o tipo das variáveis

O nome dado à uma variável deve estar de acordo com o tipo da variável. Deve-se evitar utilizar mais de um termo para se referir ao mesmo tipo de variável. Prefira `getOrder() (Order, error)` à variações desnecessárias como `getOrderInfo() (Order, error)` ou `getOrderData() (Order, error)`.

2.1.1.3 Uso de nomes procuráveis e pronunciáveis

Trata-se de substituir constantes por variáveis com nomes significativos e que possam ser procuradas no código usando ferramentas de busca.

```
1 worker.Schedule(86400000)
```

Neste exemplo, podemos observar que é impossível inferir o que significa o número passado como parâmetro para a função `Schedule`. Ao declarar uma constante com um nome significativo, podemos mitigar este problema facilitando a leitura e o entendimento:

```
1 const MillisecondsInADay int = 24 * 60 * 60 * 1000
2 worker.Schedule(MillisecondsInADay)
```

Além disso, facilitamos a busca destas informações utilizando ferramentas de busca dos softwares e IDEs utilizadas para codificação.

2.1.1.4 Não repetir contexto

Não há necessidade de repetir o contexto no nome de uma variável que já pertence ao contexto em questão. Exemplo de repetição de contexto:

```
1 type User struct {
2     UserID    uint
3     UserEmail string
4 }
```

Exemplo sem repetição de contexto:

```
1 type User struct {
2     ID    uint
3     Email string
4 }
```

2.1.2 Funções

Como nas variáveis, funções costumam ter nome negligenciados. Entretanto, no caso das funções, existem diversos outros problemas mais graves que podem ser mitigados seguindo as diretrizes do Código Limpo.

2.1.2.1 Argumentos e responsabilidade única

Deve-se reduzir ao máximo o número de argumentos passados para funções. Se uma função recebe mais de três argumentos, provavelmente tem mais responsabilidades do que deveria, infringido o Princípio da Responsabilidade Única do SOLID, descrito em detalhes na sessão Conceitos Básicos. Além disso, ter diversos argumentos resulta em uma explosão combinatória de casos de teste (testes automatizados são descritos na sessão Conceitos Básicos).

2.1.2.2 Nomes de funções

Assim como as variáveis, funções devem ter nomes que descrevem ação sendo executada usando *mixed caps* (PascalCase, camelCase, snake_case, etc.). Ainda que o nome fique grande, a facilidade de entendimento do código é bastante beneficiada.

Evite:

```
dupLS(LS *LinSist_t) (*LinSist_t, error)
```

Prefira, ao invés:

```
duplicateLinearSystem(
    linearSystem *LinearSystem,
) (*LinearSystem, error)
```

2.1.2.3 Código duplicado

Duplicação de código significa que, durante uma manutenção, diversos lugares precisam ser modificados, dificultando o processo e levando à problemas imprevistos. Pode-se evitar duplicação criando boas abstrações que podem ser reutilizadas em diversos contextos diferentes.

2.1.2.4 Uso de flags *como parâmetros*

O termo *flag* é utilizado para se referir à variáveis do tipo booleano (verdadeiro ou falso). O uso de *flags* como parâmetro na assinatura indica que a função tem mais de uma responsabilidade. Nestes casos, a função deve ser divida em duas, com responsabilidade única. Exemplo de função utilizando *flag* como argumento:

```
func (fileReader *FileReader) ReadFile(
    path string,
    temp bool,
) ([]byte, error) {
    if temp {
        return ioutil.ReadFile("/temp/"+path)
    }
    return ioutil.ReadFile(path)
}
```

Separando em duas funções de responsabilidade única:

```
func (fileReader *FileReader) ReadFile(path string) ([]byte, error) {
    return ioutil.ReadFile(path)
}

func (fileReader *FileReader) ReadTempFile(path string) ([]byte, error) {
    return ioutil.ReadFile("/temp/"+path)
}
```

2.1.3 Evitando efeitos colaterais

Uma função produz um efeito colateral se faz qualquer coisa além de receber um valor e retornar outro valor. Efeitos colaterais como modificar o valor de uma variável fora do escopo,

persistir algum dado, ou modificar o estado de uma classe, devem ser evitados. Entretanto, em momentos onde efeitos colaterais são de fato necessários, devem ser centralizados em um — e apenas um — lugar, como um serviço de armazenamento em banco de dados, por exemplo.

2.1.4 Priorizar composição à herança

Herança é sobre desenhar objetos de acordo com o que eles são, enquanto que composição trata-se de desenhar objetos de acordo com o que eles fazem. O desenho de um sistema usando herança pode levar a diversos problemas.

Vamos estabelecer que um sistema hipotético possui as classes `Analyst` (analista), com o método `punchClock` (bater ponto); e Manager (gestor), com o método `approveTimeTrackingRecord` (aprovar folha de ponto). Como ambos são colaboradores da mesma empresa, ambos possuem o método `calculateSalary` (calcular salário). A fim de não duplicar o método `calculateSalary`, um arquiteto poderia desenhar uma classe `Employee` (funcionário), e tanto `Analyst` quanto Manager poderiam herdar o método `calculateSalary` de `Employee`. Adiciona-se, então, a classe `Director` (diretor), com o método `generateReport` (gerar relatório), e classe `VicePresident` (vice presidente), com o método `approveReport` (aprovar relatório). Ambos possuem o método `approveBudget` (aprovar orçamento), e portanto é criada uma classe `Executive` (executivo), e assim ambos podem herdar o método `approveBudget`.

Com o tempo a empresa cresce, e surgem novos requisitos ao software, por exemplo, a necessidade de criar uma representação para a figura do CEO (*Chief Executive Officer*), que é pago via pró-labore, deve poder aprovar orçamentos (`approveBudget`), aprovar relatórios (`approveReport`) e aprovar a folha de ponto dos analistas na ausência de seus gestores (`approveTimeTrackingRecord`). Naturalmente, o arquiteto poderia construir uma classe CEO que herda de `VicePresident` (para ter acesso ao método `approveReport` e ao método `approveBudget`, herdado de `Executive`) e de Manager (para ter acesso ao método `approveTimeTrackingRecord`). A ilustração deste cenário pode ser observada na figura 2.1 Desta forma, o CEO "é" um executivo, um diretor, e um funcionário ao mesmo tempo, gerando uma inconsistência na representação da realidade em objetos. Além disso, a classe CEO agora possui o método `calculateSalary` herdado de `Employee`, porém não utiliza sua funcionalidade, já que em nossa empresa hipotética o CEO é pago através de pró-labore.

Existem outras maneiras de desenhar o CEO neste caso, mas todas são igualmente problemáticas, gerando confusão na representação e/ou adição de funcionalidades desnecessárias em outras classes.

Como o software muda ao longo do tempo de acordo com as novas funcionalidades e requisitos, situações semelhantes a essa são comuns. A alternativa à este desenho é utilizar composição: uma maneira de "compor"classes utilizando outras classes, não herdando delas. É possível conceber qualquer relação de herança (fulano "é" ciclano) através de uma relação de

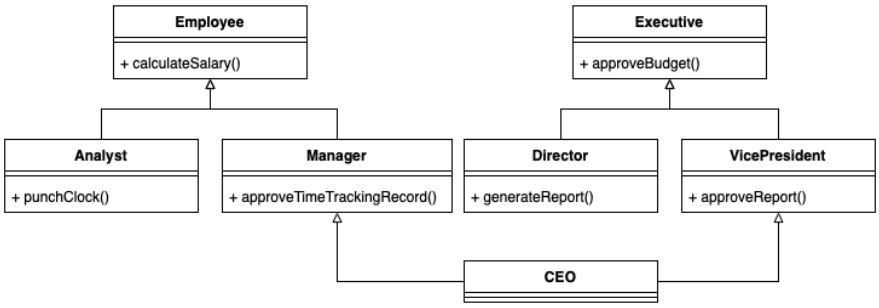

Figura 2.1: Diagrama de Classes (simplificado) representando a relação de herança. Fonte: o autor.

composição (fulano “tem"ciclano). Desta forma, podemos definir nossas classes da seguinte maneira:

- Um analista é um “batedor de ponto” e um “recebedor de salários”;
- um gestor é um “aprovador de folhas de ponto” e um “recebedor de salários”;
- um diretor é um “gerador de relatórios” e um “aprovador de orçamentos”;
- um vice presidente é um “aprovador de relatórios” e um “aprovador de orçamentos”.

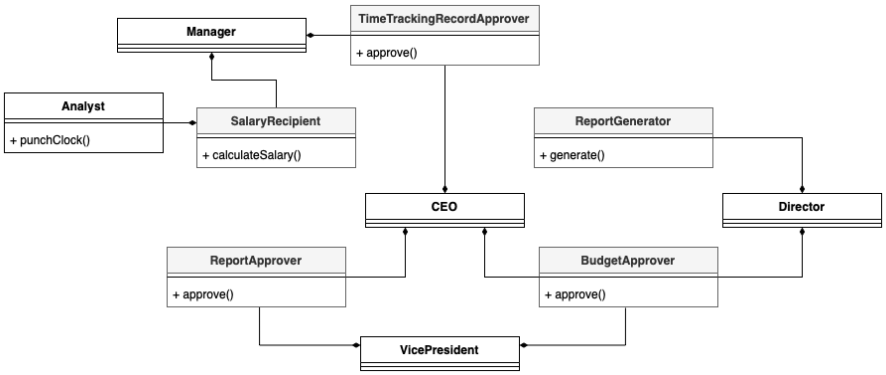

Figura 2.2: Diagrama de Classes (simplificado) representando a relação de composição. Fonte: o autor.

A ilustração do cenário de composição pode ser observada na figura 2.2. Assim, as classes são desenhadas de acordo com o que elas fazem, e não de acordo com o que elas são. Obtemos então um desenho desacoplado e orientado à funcionalidades, e portanto podemos facilmente definir o CEO como “aprovador de orçamentos”, "aprovador de relatórios” e “aprovador de folhas de ponto”. Podemos dizer, então, que a herança não é um bom padrão de projeto, pois dificulta a evolução à longo prazo, (apesar de existirem — raros — casos onde pode ser aplicada de maneira eficaz), já que obriga o arquiteto a “prever o futuro” à medida que desenha a taxonomia de objetos em um estágio bastante inicial do projeto. Desta forma é possível concluir que é uma boa prática preferir sempre composição à herança.

2.2 ARQUITETURA LIMPA

O termo tornou-se conhecido no livro Arquitetura Limpa, O Guia do Artesão para Estrutura e Design de Software, de Robert C. Martin (Martin, 2019), e, como o Código Limpo, tornou-se muito popular na comunidade de engenheiros de software. A obra indica escolhas a serem feitas ao planejar a arquitetura de um sistema e explica detalhadamente o motivo destas escolhas serem importantes para o sucesso do projeto, baseado em mais de cinquenta anos de experiência dos autores na área.

Ao longo dos anos foram desenvolvidos diversos padrões de design de sistemas em camadas, como a Arquitetura Hexagonal (também conhecida como arquitetura de Portas e Adaptadores) (Freeman, 2009), arquitetura "Cebola"(*Onion Architecture*) (Palermo, 2008), *Screaming Architecture* (Martin, 2011), a BCE (Jacobson, 1992), entre outras. Todas estas, entretanto, tem em comum a separação das responsabilidades ou “preocupações”, o que é feito através da separação do sistema em camadas, isolando as regras de negócio das interfaces.

Estes diversos tipos de arquitetura diferentes tem como principal objetivo organizar os sistemas de forma que sejam independentes de bibliotecas externas, sejam testáveis (cada componente, de forma isolada) e não dependam de tecnologias específicas (ex.: banco de dados, cache, *message broker*, etc.).

O conjunto de práticas descritas no livro vai desde paradigmas de linguagens de programação até uma descrição detalhada dos princípios SOLID, passando por temas como Orientação à Objetos, coesão, acoplamento, “componentização”, orientação à interfaces, entre outros temas.

2.3 REDUÇÃO DE COMPLEXIDADE CICLOMÁTICA

Criada por Thomas J. McCabe (McCabe, 1976), a Complexidade Ciclomática é uma métrica para medir a complexidade de um módulo, função ou classe com base no número de decisões (condicionais) presentes nos fluxos de execução. Quanto mais condicionais um programa tem, mais caminhos possíveis existem para a execução, e mais difícil é de entender, manter e evoluir o código. Além disso, quanto maior a complexidade ciclomática, mais casos de teste unitário precisam ser escritos.

Formalmente, a Complexidade Ciclomática de um trecho de código é definida pelo número caminhos linearmente independentes possíveis. Por exemplo, se o código não possui condicionais, a complexidade é 1; se o código tem apenas uma condicional, então há dois caminhos, um para cada caso da condicional, ou seja, a complexidade ciclomática é 2. Duas condicionais (uma dentro da outra) ou uma condicional com duas comparações, indica complexidade ciclomática igual a 3.

Existem diversos motivos para evitar que a Complexidade Ciclomática seja grande: maior carga cognitiva (mapeamento mental de fluxos) dificultando a leitura e o entendimento do código; maior chance de ter imprevisibilidade na execução das instruções; maior chance

de ter *bugs*, e crescente dificuldade em adicionar funcionalidades por conta da quantidade de fluxos possíveis; e maior número de casos de teste unitário necessários para cobrir todas as possibilidades do código.

Considere o seguinte cenário: um *e-commerce* possui uma estrutura de dados `Order` para representar um pedido, que possui uma lista de produtos, representados por uma estrutura de dados chamada `Product`. O código da função `CalculatePrice`, que calcula o preço total de um pedido, pode ser representado pelo fluxograma ilustrado na figura 2.3:

```
1  func (order *Order) CalculatePrice() (uint64, error) {
2      if len(order.Products) == 0 {
3          return 0, errors.New("order has no products")
4      }
5      price := uint64(0)
6      for _, product := range order.Products {
7          price += product.Price * uint64(product.Amount)
8          if !product.HasFreeShipping(order.Customer.Zipcode) {
9              price += product.CalculateShippingPrice(
10                 order.Customer.Zipcode
11             )
12             if product.DeliveredByShippingCompany {
13                 price += ShippingCompanyTax
14             }
15         }
16     }
17     return price, nil
18 }
```

A Complexidade Ciclomática M é dada pela formula M = E – N + 2P, onde,

- E = o número de arestas no fluxograma;
- N = o número de nós no fluxograma;
- P = o número de componentes conectados no fluxograma.

Portanto, a Complexidade Ciclomática da função `CalculatePrice` é M = 5. Ao longo do tempo, tornou-se consenso tanto na indústria como nos materiais acadêmicos que:

- Se $1 \leq M \leq 10$, o procedimento é simples e possui pouco risco;
- se $11 \leq M \leq 20$, o procedimento possui complexidade média com risco moderado;
- se $21 \leq M \leq 50$, o procedimento é complexo e de alto risco; e
- se $M < 50$, não é possível testar o código e o risco é altíssimo.

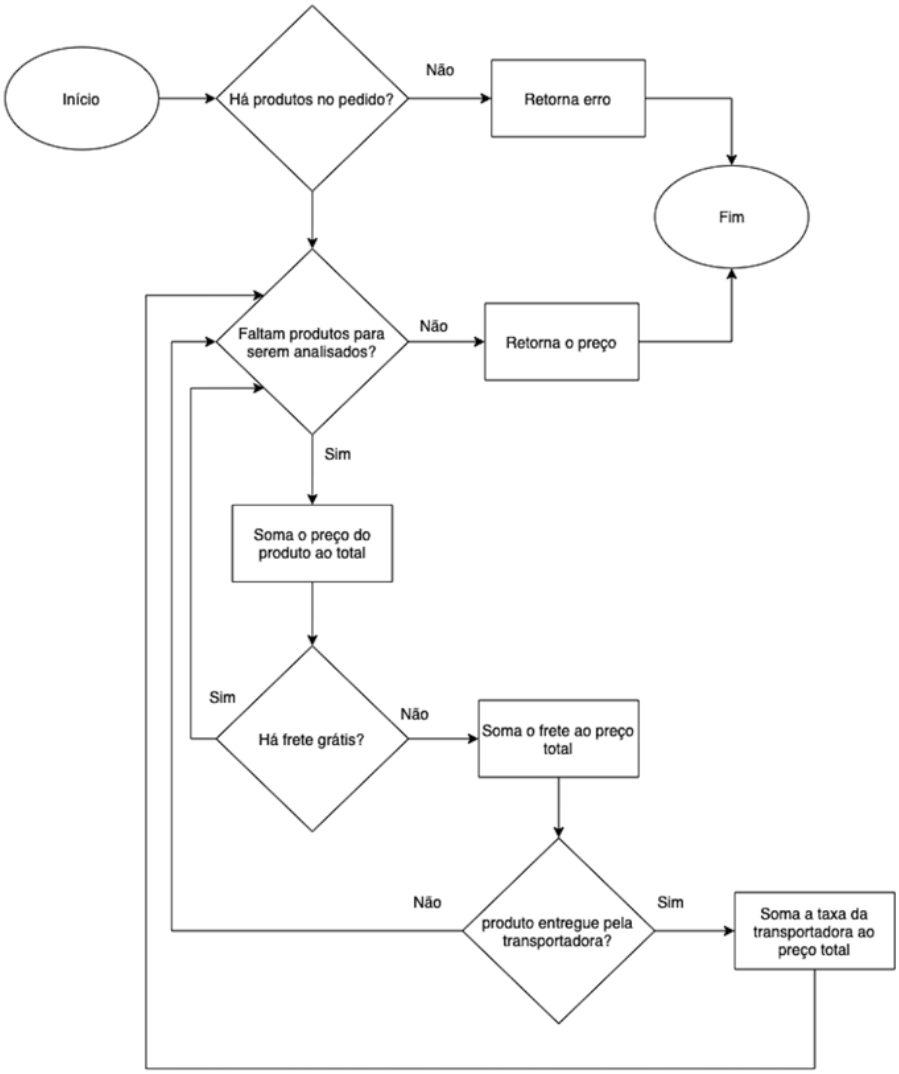

Figura 2.3: Fluxograma da função `CalculatePrice`. Fonte: o autor.

Neste exemplo, podemos utilizar o Princípio da Responsabilidade Única (descrito na sessão Conceitos Básicos) e o princípio "*Tell Don't Ask*" (Fowler, 2013) para reduzir a complexidade ciclomática da função terceirizando as responsabilidades de verificar se há frete grátis e se devemos somar a taxa da transportadora para o método `CalculateShippingPrice` da entidade `Product`. Reestruturando o código, obtemos:

```
1  func (order *Order) CalculatePrice() (uint64, error) {
2      if len(order.Products) == 0 {
3          return 0, errors.New("order has no products")
4      }
5      price := uint64(0)
6      for _, product := range order.Products {
7          price += product.Price * uint64(product.Amount)
8          price += product.CalculateShippingPrice(order.Customer.Zipcode)
9      }
10     return price, nil
11 }
```

Neste cenário, a função `CalculateShippingPrice` retorna zero caso não haja nenhum valor a ser acrescentado.

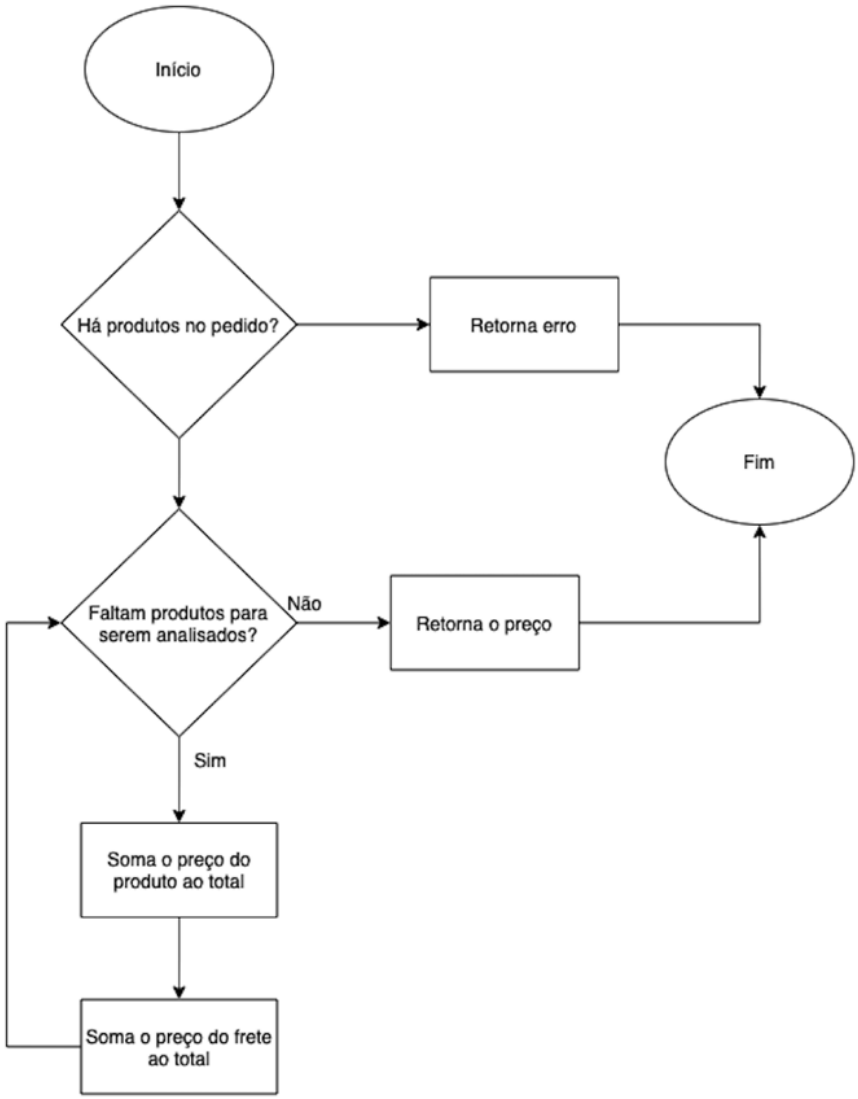

Figura 2.4: Fluxograma da função `CalculatePrice` depois da reestruturação. Fonte: o autor.

A complexidade ciclomática M' após a reestruturação é 3. A palavra "complexo" pode ser definida como "que abarca e compreende vários elementos ou aspectos distintos apresentando relações de interdependência" (de Português, 2021), portanto, para além do fato de que M' < M, podemos observar que o fluxograma da função representado na figura 2.4 possui uma quantidade menor de nós e arestas, e é, portanto, menos complexo que o fluxograma representado na figura 2.3. Reduzir a Complexidade Ciclomática de um trecho de código torna o programa mais fácil de ser lido e entendido, favorecendo a rápida manutenção e adição de novas funcionalidades se necessário.

No dia a dia de desenvolvimento, entretanto, não costuma ser necessário fazer o cálculo da Complexidade Ciclomática. Basta estar ciente de que é necessário reduzi-la sempre que possível, e o código naturalmente passa a ser cada vez menos complexo.

2.4 DESENVOLVIMENTO ORIENTADO À TESTES

Teste de software é uma disciplina extensa e possui diversas abordagens. Uma maneira eficiente de testar software é desenvolver testes automatizados, literalmente programas que testam outros programas. Esta técnica é bastante eficaz porque à medida que o software muda ao longo do tempo, os testes podem ser repetidamente executados ao longo do processo de desenvolvimento a fim de verificar se as mudanças inseriram problemas inesperados nas regras de negócio. Além disso, os testes automatizados podem ser inseridos na esteira de CI/CD (*Continuous Integration and Continuous Deleivery*), garantindo que o software sendo entregue em produção funciona como o esperado, atende os requisitos, e não vai prejudicar os fluxos atuais. Por isso, desenvolver testes automatizados tem se tornado não apenas uma prática recomendada, mas muitas vezes obrigatória em diversas empresas de tecnologia.

O Desenvolvimento Orientado à Testes (TDD, ou *Test Driven Development*) consiste em desenvolver testes automatizados para os requisitos do software antes mesmo que as funcionalidades sejam de fato desenvolvidas. O principal benefício desta prática reside em “forçar"o desenvolvedor a pensar na implementação e nas interfaces da funcionalidade antes de iniciar o desenvolvimento, o que gera ideias de como melhorar a implementação, ajuda a excluir desenhos ineficientes (do ponto de vista de arquitetura/design de software) da lista de possibilidades, e clarifica as regras de negócio. Além disso, o TDD ajuda a fixar a lógica dos trechos testados e garante que ela não mude — sem antes fazer com que o teste falhe — , garantindo que trechos do código funcionam corretamente mesmo após modificações e que os requisitos estão sendo cumpridos corretamente. Além disso, os testes também funcionam como uma espécie de "documentação executável” do trecho testado, onde fica fácil entender o comportamento da funcionalidade para desenvolvedores que acabaram de ingressar ao projeto, precisam entender alguma regra de negócio específica, ou até mesmo o próprio desenvolvedor que escreveu a funcionalidade à muito tempo e já não lembra como funciona mais.

Desenvolver usando TDD geralmente inclui diversos tipos de testes automatizados, os mais comuns sendo:

- Teste unitário: testa uma única função de forma isolada, geralmente utilizando *mocks* (componentes falsos que imitam componentes reais, apenas para simular o comportamento de uma dependência externa ao código que está sendo testado de forma unitária) quando necessário;
- Teste de integração: testa mais de um componente ao mesmo tempo, também admite o uso de *mocks*;

- Teste de ponta a ponta: testa fluxos completos do software de uma ponta à outra (ex.: execução de *scripts* automatizados na interface de usuário, seguido de asserção do conteúdo persistido no banco de dados);

Existem diversos outros tipos de testes automatizados, e a nomenclatura utilizada varia bastante de literatura para literatura, e de empresa para empresa. Entretanto, a grande maioria pode ser encaixada em um destes três tipos.

2.5 REVISÃO DE CÓDIGO

Revisão de código, ou *Code Review*, trate-se da prática de ler com atenção as modificações propostas ao software antes de integra-las à base de código. Uma prática comum, e recomendada, é que a revisão seja feita por mais de um desenvolvedor, de preferência que não tenham trabalhado no desenvolvimento da modificação em questão. A revisão de código é fundamental para buscar possíveis problemas que passaram despercebidos pelo desenvolvedor da funcionalidade, garantir o cumprimento das boas práticas estabelecidas pela organização, bem como buscar maneiras de melhorar a eficiência e o design da nova funcionalidade a fim de garantir a fácil manutenção no longo prazo. A prática geralmente é acompanhada de um *checklist* de pontos de atenção que o revisor deve atentar-se para garantir que o código está pronto para ser integrado.

Abaixo, está o exemplo de um *checklist* de *code review* de um dos times de engenharia de uma grande empresa de tecnologia do Brasil:

- O código segue as boas práticas definidas pelo *Clean Code*;
- A lógica implementada está correta e segue a especificação descrita na tarefa;
- A complexidade ciclomática do código é a menor possível;
- A complexidade algorítmica do código é a menor possível;
- Todos os casos da lógica implementada estão cobertos por teste unitários e/ou de integração, inclusive *corner cases*;
- O código utiliza imutabilidade onde é possível e prático utilizar;
- Não há código duplicado;
- A estrutura das modificações está desacoplada e orientada à interfaces;
- Não há erros de gramática em nomes, comentários e descrições de testes;
- Todo o código (inclusive comentários) está em inglês;
- A descrição de cada caso de teste está compreensível e faz sentido com o que está sendo testado;

- O *linter* da linguagem não acusa nenhum problema;
- O código obedece a convenção de nomes da linguagem (variáveis, *packages*, módulos, funções, etc.)
- A quantidade de linhas modificadas na *Pull Request* está ao redor de 200;
- Não existem brechas de segurança nas modificações propostas;
- Os erros/exceções estão sendo corretamente tratados(as);
- Não há comentários desnecessários no código;
- Não há trechos de código não utilizados;
- Funcionalidades ainda não finalizadas estão protegidas por *feature toggles* ou alguma outra ferramenta equivalente;
- As modificações seguem o case padrão do projeto (camelCase, PascalCase, snake_case, etc);
- Não há *logs* desnecessários, e todos os *logs* seguem o padrão do RFC estabelecido;

Comumente, se o código segue todos os tópicos destacados no *checklist* e os revisores não possuem nenhuma sugestão de melhoria, as modificações são aprovadas e seguem para a fase de integração com o resto da base de código. Caso haja alguma sugestão ou modificação proposta, é uma boa prática gerar discussão ao redor das propostas e chegar a consenso coletivo sobre o que mudar, e porque mudar. Desta forma, além de ajudar a garantir a qualidade do código, a prática do *Code Review* torna-se uma poderosa ferramenta para ensinar, documentar e compartilhar aprendizados

2.6 INTEGRAÇÃO CONTÍNUA E ENTREGA CONTÍNUA

Integração contínua trata-se da prática de integrar novos trechos de código à base de código frequentemente — várias vezes ao dia —, enquanto entrega contínua refere-se à prática entregar software em produção frequentemente — também, várias vezes ao dia, contrapondo-se ao processo de ciclos grandes de desenvolvimento que resultam em *releases* com diversas mudanças, que são todas integradas e entregues de uma vez só. Esta é considerada uma boa prática porque proporciona um substancial aumento de velocidade no desenvolvimento, além de garantir que os usuários do software estão sempre em contato com as novas funcionalidades, gerando um ciclo de *feedback* contínuo que ajuda a direcionar os requisitos para as necessidades reais dos usuário. Por isso, a prática do chamado CI/CD tornou-se bastante popular nas empresas de tecnologia, especialmente nas que lidam com produtos inovadores, por conta do constante *feedback* do usuário.

Entretanto, para conseguir integrar e entregar software de forma contínua é necessário um esforço quanto à automatização das etapas de integração e entrega, bem como instituição de uma grande porcentagem de cobertura da base de código por testes automatizados — idealmente 100% —-, a fim de que as contínuas entregas não prejudiquem o software em funcionamento e não atrasem a equipe, que do contrário teria que se preocupar em fazer o processo de forma manual várias vezes ao dia.

Existem diversas ferramentas para desenvolvimento de esteiras CI/CD, que tratam de automatizar todas as partes do processo para que o código seja integrado e entregue de forma eficiente e sem problemas, e as equipes precisam de maturidade em seus processos para atingir este fluxo de desenvolvimento. Por isso, utilizar o processo de integração e entrega contínua é considerado uma boa prática.

2.7 DOCUMENTAÇÃO DE SOFTWARE

Existem diversas maneiras de documentar software, entretanto, o fato de documentar, independente do processo seguido, é considerado uma boa prática porque garante que os colaboradores que não estão envolvidos no desenvolvimento — e, portanto, não conhecem os detalhes de implementação e regras de negócio — possam entender de forma rápida os detalhes necessários para executar qualquer tipo de ação relacionada ao software, seja inserir uma nova funcionalidade, executar uma manutenção, ou até integrar com outro software e etc. A documentação pode ser feita desde os estágios iniciais do projeto em forma de levantamento de requisitos — que documenta o funcionamento do software em detalhes e pode ser utilizado para descrever tarefas mais específicas, ou até mesmo ser anexado à um contrato de desenvolvimento — até documentação dos *endpoints* de uma API, por exemplo, para facilitar a integração com outros softwares. A documentação pode inclusive adentrar a própria base de código do projeto, por exemplo, utilizando comentários (funcionalidade disponível na maioria das linguagens de programação) para alertar outros desenvolvedores sobre idiossincrasias relacionadas ao trecho comentado.

Documentar software é um esforço pequeno e que economiza muito tempo no longo prazo, já que reduz a necessidade de contato de terceiros com a equipe de desenvolvimento e acelera o *onboarding* de novos desenvolvedores, e portanto é considerada uma prática fundamental.

2.8 PRINCÍPIOS SOLID

SOLID é um acrônimo para cinco princípios da engenharia de software cujo objetivo é melhorar o design de software a fim de que seja flexível, fácil de entender e fácil de executar manutenções. Este princípios foram introduzidos por Robert C. Martin no artigo *Design Principles and Design Patterns* (Martin, 2002). Abaixo estão listados, em tradução do autor, os cinco princípios SOLID:

- **Single responsibility principle (princípio da responsabilidade única)**: Não deve haver mais de uma razão para que uma classe mude. Em outras palavras, cada classe deve ter apenas uma responsabilidade.
- **Open-closed principle (princípio do aberto e fechado)**: componentes do software devem estar abertos para extensão, porém fechados para modificação.
- **Liskov substitution principle (princípio da substituição de Liskov)**: funções que usam ponteiros ou referencias para classes base precisam poder utilizar objetos de classes derivadas sem conhecê-las.
- **Interface segregation principle (princípio da segregação de interfaces)**: muitas interfaces específicas para clientes são melhores do que uma interface de propósito geral.
- **Dependency inversion principle (princípio da inversão de dependência)**: componentes devem depender de abstrações, não de implementações concretas.

Existe ampla literatura sobre cada um destes princípios, mas é importante ressaltar que conhece-los e segui-los aumenta a chance de escrever códigos flexíveis o suficiente para que a manutenção e adição de novas funcionalidades seja mais fácil, e, portanto, tome menos tempo, consolidando o SOLID como uma boa prática de engenharia.

2.9 METODOLOGIA TWELVE-FACTOR APP

Trata-se de um conjunto de regras para desenvolvimento de softwares como serviço (*software as a service*, ou SAAS), que são cada vez mais comuns na indústria. Retirado do próprio site oficial (disponível no endereço https://12factor.net (Wiggins, 2017)), e com tradução do autor, a metodologia garante que, se seguida, ajuda a criar aplicações que:

- Use formatos declarativos para automação de configuração, para minimizar tempo e custo para novos desenvolvedores ingressando no projeto;
- Ter um contrato limpo com o sistema operacional subjacente, oferecendo máxima portabilidade entre ambientes de execução;
- São adequados para implantação em modernas plataformas em nuvem, dispensando a necessidade de administração de servidores e sistemas;
- Minimize a divergência entre desenvolvimento e produção, permitindo implantação contínua para máxima agilidade; e
- Pode escalar sem alterações significativas nas ferramentas, arquitetura ou práticas de desenvolvimento.

A metodologia de doze fatores pode ser aplicada a aplicativos escritos em qualquer linguagem de programação e que usam qualquer combinação de serviços de apoio (banco de dados, fila, cache de memória, etc.). A metodologia é composta de 12 princípios, aqui listados, com tradução do autor:

1. Base de código: uma base de código rastreada no controle de revisão, muitas implantações;
2. Dependências: declare e isole explicitamente dependências;
3. Configuração: armazenar a configuração no ambiente;
4. Serviços de apoio: trate os serviços de apoio como recursos anexados;
5. Construir, publicar, executar: estágios de compilação e execução estritamente separados;
6. Processos: execute o aplicativo como um ou mais processos sem estado;
7. Vinculação de porta: exportar serviços via ligação de porta;
8. Concorrência: escale horizontalmente por meio do modelo de processo;
9. Descartabilidade: maximize a robustez com inicialização rápida e desligamento gracioso;
10. Paridade de desenvolvimento/produção: mantenha o desenvolvimento, a preparação e a produção o mais semelhante possível;
11. Logs: devem ser tratados como *streams* de eventos;
12. Processos de administrador: execute tarefas de administração/gerenciamento como processos únicos.

Este conjunto de práticas foi desenvolvido por uma série de engenheiros experientes, e foi naturalmente adotada pela indústria como um padrão a ser seguido ao desenvolver software como serviço.

2.10 REDUÇÃO DE COMPLEXIDADE ALGORÍTMICA

A complexidade algorítmica está relacionado à quantidade de tempo que um computador leva para executar um algoritmo, comumente estimada através da contagem de operação elementares que o processador executa, e expressada através da notação *big O* ($O(n)$, $O(n \log n)$, $O(n2)$, etc., onde n refere-se ao tamanho da entrada).

A redução de complexidade algorítmica pode garantir que o trecho de código execute mais rápido, economizado recursos computacionais — que é especialmente importante diante de um cenário em que a computação em nuvem, cobrada por demanda, está cada vez mais comum

— e melhorando a experiência do usuário. Além disso, reduzir a complexidade algorítmica frequentemente também simplifica o trecho, garantindo mais facilidade de entendimento e manutenção, e por isso é considerada uma boa prática.

2.11 AUMENTO DE COESÃO E REDUÇÃO DE ACOPLAMENTO

A coesão é relacionada à o quanto os componentes de uma classe estão relacionados uns aos outros. Uma classe é dita com baixa coesão quando, por exemplo, possui grupos de atributos que são utilizados por grupos de métodos, porém não pela maioria destes métodos, indicando que provavelmente a classe possui mais de uma responsabilidade. Acoplamento, por sua vez, refere-se ao quanto duas classes dependem uma da outra. Duas classes são caracterizadas como pouco acopladas quando mudanças feitas em uma delas não afeta a outra. Isso pode ser atingido, principalmente, isolando as responsabilidades entre as classes e definido contratos (interfaces, por exemplo) para interações entre elas.

2.12 REDUÇÃO DE COMPLEXIDADE ESPACIAL

A complexidade espacial de um algoritmo está relacionada a quantidade de memória utilizada pelo trecho ao executar uma tarefa ou resolver um problema de acordo com as características da entrada. Reduzir a complexidade espacial pode ser especialmente importante durante a execução em ambientes com recursos limitados (aplicações mobile, por exemplo), ou que não escalam verticalmente de forma automática. Por isso, é listada como uma boa prática para a engenharia moderna.

3 METODOLOGIA

A sessão Conceitos Básicos foi redigida através de um levantamento com base em uma pesquisa quanto às boas práticas mais comumente utilizadas e evangelizadas por equipes de desenvolvimento na indústria. Entretanto, elas podem ou não ser conhecidas e aplicadas pelas equipes. A fim de executar uma investigação sobre o quão difundidas são as práticas descritas nas empresas de tecnologia do Brasil, um questionário que aborda cada uma delas foi aplicado a um grupo de 33 profissionais da área, de diversas unidades federativas. O questionários aborda três principais áreas:

1. Informações sobre o participante;
2. Perguntas relacionadas ao nível de conhecimento dos participantes quanto a cada uma das práticas listadas;
3. Perguntas relacionadas à opinião dos participantes quanto a cada uma das práticas listadas.

O questionário foi publicado utilizando a ferramenta Google Forms, e divulgado em grupos de discussão sobre tecnologia, grupos de times de engenharia de empresas de tecnologia privadas e grupos de estudantes de cursos relacionados à área de tecnologia da informação. Abaixo estão todas as questões, exatamente como foram publicadas, junto com os resultados da pesquisa. Observe que existem questões na escala de Likert. Neste caso, o número 1 representa "Não concordo"e o número 5 representa "Concordo".

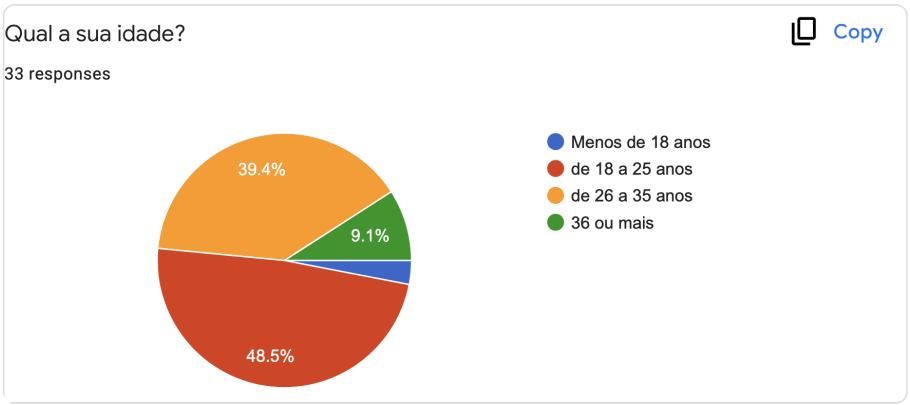

Figura 3.1: Pergunta: "Qual a sua idade?". Fonte: Google Forms.

O objetivo das perguntas cujos resultados estão demonstrados nas figuras 3.1 e 3.2 é entender se existe alguma relação entre o tempo de experiência dos participantes e o fato de conhecerem ou não as práticas apresentadas, bem como a aplicação ou não das mesmas em seus respectivos projetos.

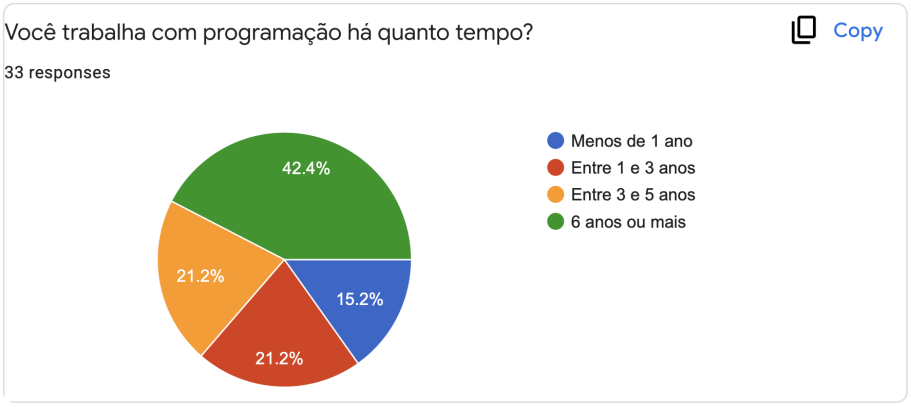

Figura 3.2: Pergunta: "Você trabalha com programação há quanto tempo?". Fonte: Google Forms.

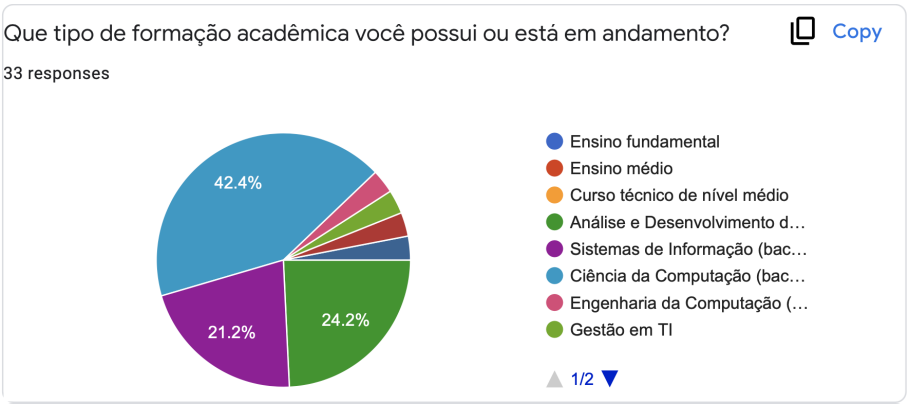

Figura 3.3: Pergunta: "Que tipo de formação acadêmica você possui?". Fonte: Google Forms.

A intenção por trás das perguntas ilustradas nas figuras 3.3 e 3.4 é entender melhor o demográfico dos participantes quanto à formação acadêmica, verificando se existe alguma relação entre a formação e a qualidade do software desenvolvido, tanto relacionado ao curso em específico quanto à natureza da instituição (pública ou privada). Não foi possível, entretanto, perceber uma diferença significativa entre as respostas de participantes de diferentes cursos ou tipos de instituição.

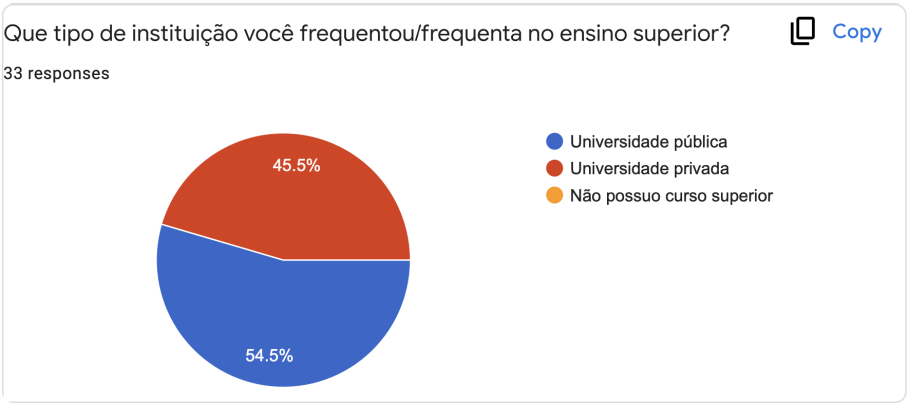

Figura 3.4: Pergunta: "Que tipo de instituição você frequentou/frequenta no ensino superior?". Fonte: Google Forms.

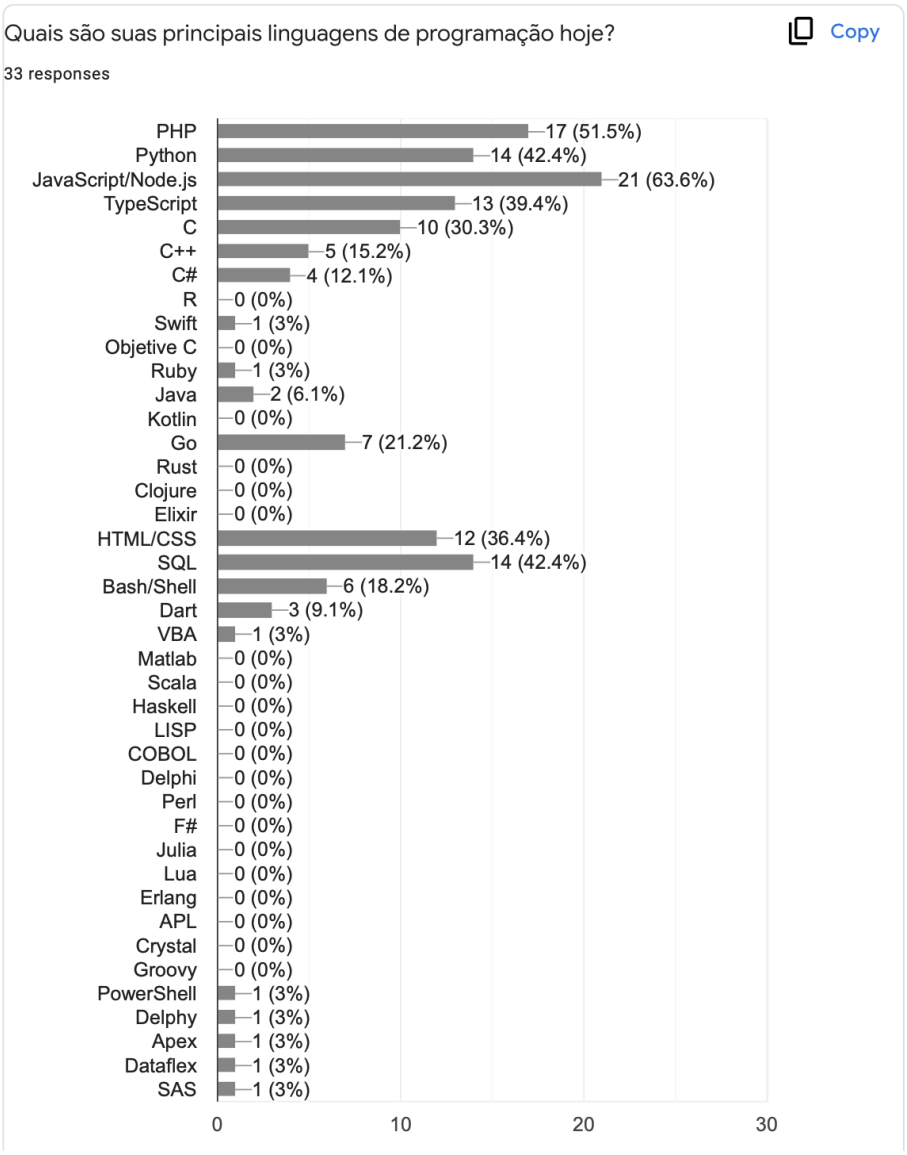

Figura 3.5: Pergunta: "Quais são suas principais linguagens de programação?". Fonte: Google Forms.

A pergunta ilustrada na figura 3.5 dedica-se a listar as linguagens de programação mais utilizadas pelos participantes a fim de ajudar a estabelecer alguma relação entre as funcionalidades da linguagem e a observação ou não das práticas listadas. Isso é importante porque existem linguagens mais ou menos permissivas, que facilitam ou dificultam a não observação de certas práticas. Além disso, é possível, ainda, estabelecer uma relação entre a linguagem e o quão atentos são os desenvolvedores que a utilizam quanto às boas práticas.

As figuras de 3.6 à 3.16 dedicam-se a entender o nível de conhecimento e prática dos participantes quanto à cada um dos conceitos apresentados. Tratam-se de perguntas relacionadas ao objetivo de entender o estado atual da difusão destas práticas nas empresas de tecnologia do Brasil, servindo também ao propósito de estabelecer relações com resultados de outras perguntas, tema mais explorado na sessão Resultados e discussão.

A pergunta ilustrada na figura 3.17 tem como objetivo estabelecer um panorama mais geral quanto ao nível de conhecimento dos participantes sobre todos os termos apresentados.

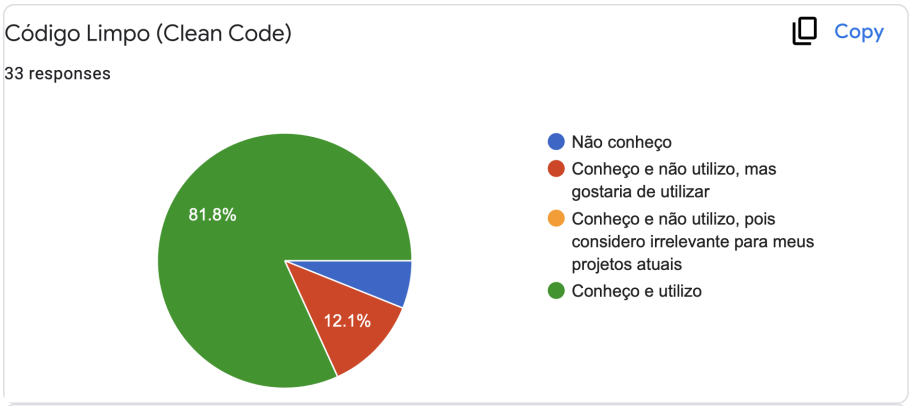

Figura 3.6: Pergunta: "Código Limpo (*Clean Code*)". Fonte: Google Forms.

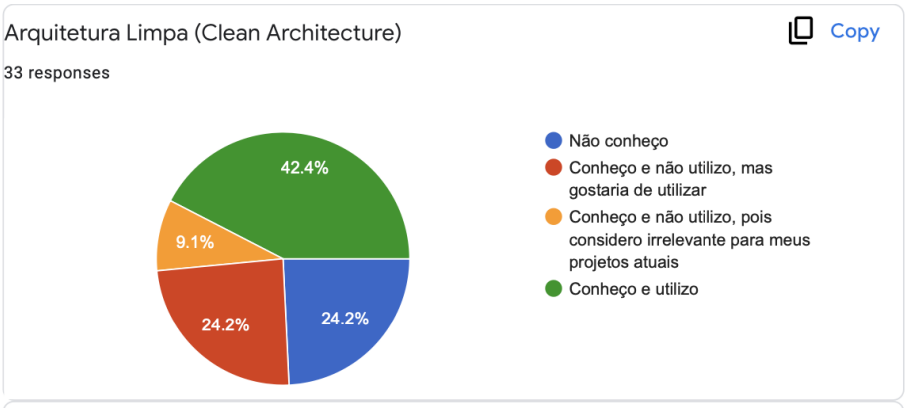

Figura 3.7: Pergunta: "Arquitetura limpa". Fonte: Google Forms.

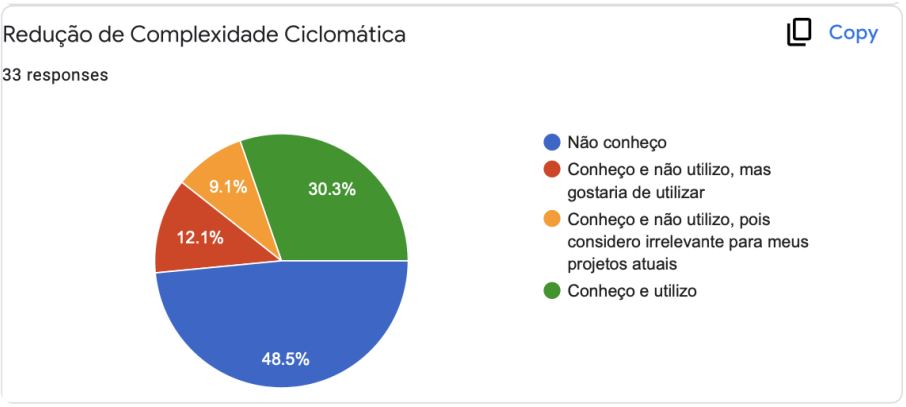

Figura 3.8: Pergunta: "Redução de Complexidade Ciclomática". Fonte: Google Forms.

Este resultado carece, entretanto, dos resultados das perguntas anteriores para que seja feita uma análise mais assertiva.

As perguntas demonstradas nas figuras 3.18 e 3.19 visam verificar se os participantes acreditam na comum falácia de que a aplicação das práticas citadas aumenta o tempo total de desenvolvimento dos projetos de software. Estas perguntas são de crucial importância para a análise apresentada na sessão Resultados e discussão, na medida em que são comparadas com a

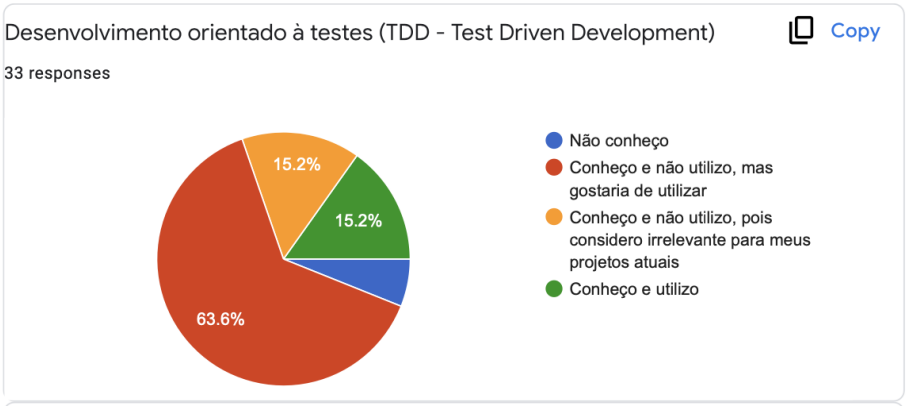

Figura 3.9: Pergunta: "Desenvolvimento orientado à testes (TDD - *Test Driven Development*)". Fonte: Google Forms.

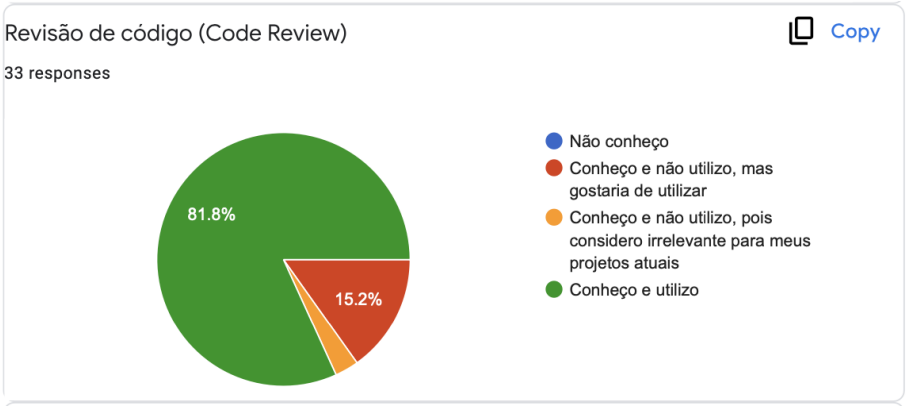

Figura 3.10: Pergunta: "Revisão de Código (TDD - *Code Review*)". Fonte: Google Forms.

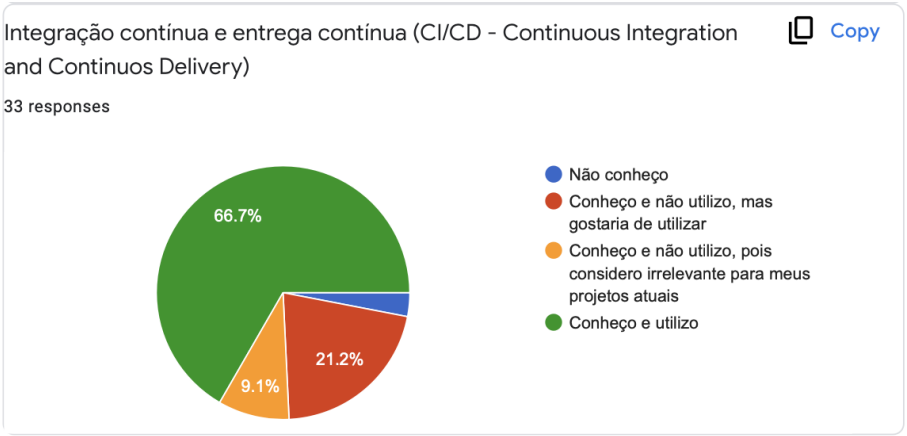

Figura 3.11: Pergunta: "Integração Contínua e Entrega Contínua (CI/CD - *Continuous Integration and Continuous Delivery*)". Fonte: Google Forms.

pergunta ilustrada na figura 3.23, a fim de buscar uma resposta assertiva quanto aos motivos da frequente não observação das práticas.

Uma das hipóteses levantadas ao elaborar o questionário seria de que os desenvolvedores talvez não enxergassem os benefícios das boas práticas comumente defendidas no âmbito da engenharia de software. A pergunta cujos resultados estão ilustrados na figura 3.20 visa verificar a

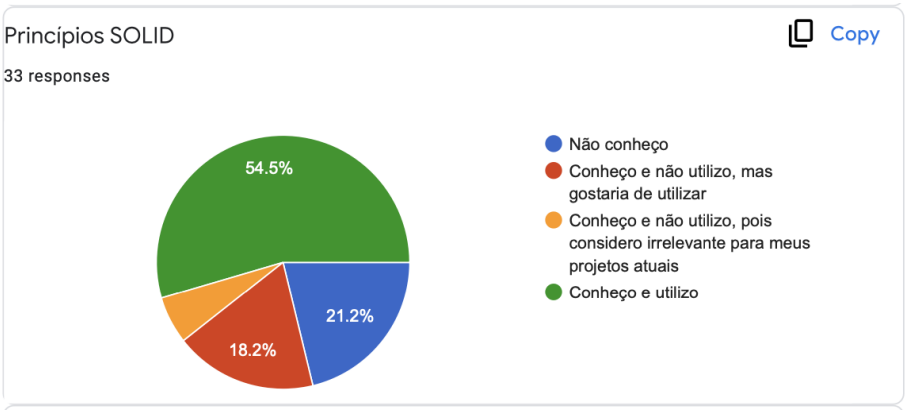

Figura 3.12: Pergunta: "Princípios SOLID". Fonte: Google Forms.

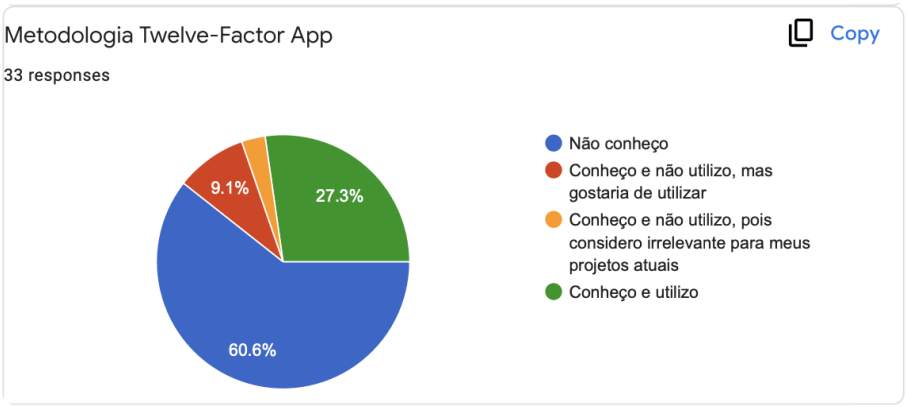

Figura 3.13: Pergunta: "Metodologia *Twelve-factor App*". Fonte: Google Forms.

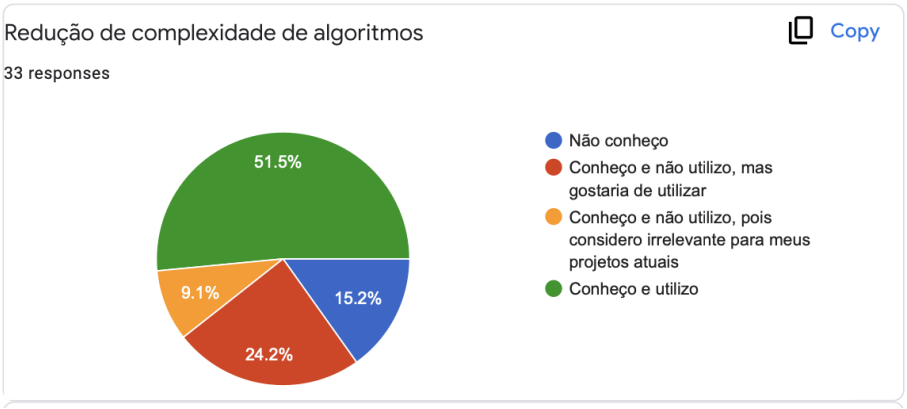

Figura 3.14: Pergunta: "Redução de complexidade de algoritmos". Fonte: Google Forms.

veracidade desta hipótese, confirmando que a grande maioria de fato concorda com os benefícios das práticas, ajudando a direcionar a investigação sobre os motivos da frequente não aplicação dos conceitos para outras hipóteses.

As questões demonstradas nas figuras 3.21, 3.22 e 3.23 tem como objetivo ajudar a elucidar a visão dos participantes quanto à falta de tempo como motivador principal da não observação das práticas durante o desenvolvimento. É interessante observar que, para que uma análise assertiva possa ser feita, a figura 3.23 é especialmente importante, já que elucida a

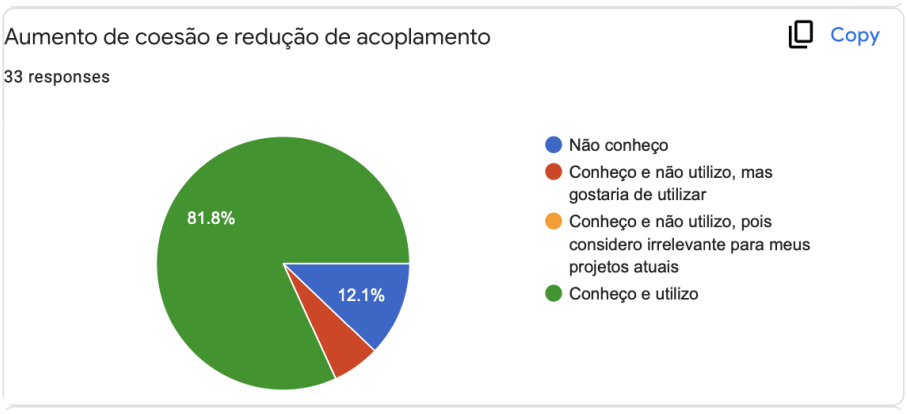

Figura 3.15: Pergunta: "Aumento de coesão e redução de acoplamento". Fonte: Google Forms.

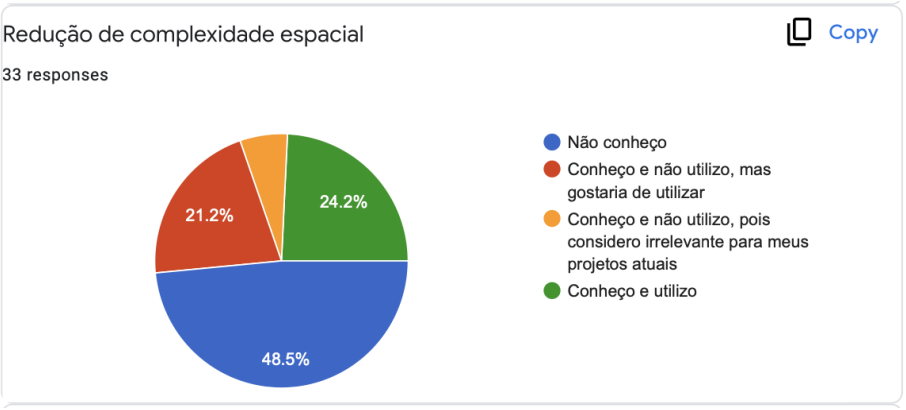

Figura 3.16: Pergunta: "Redução de complexidade espacial". Fonte: Google Forms.

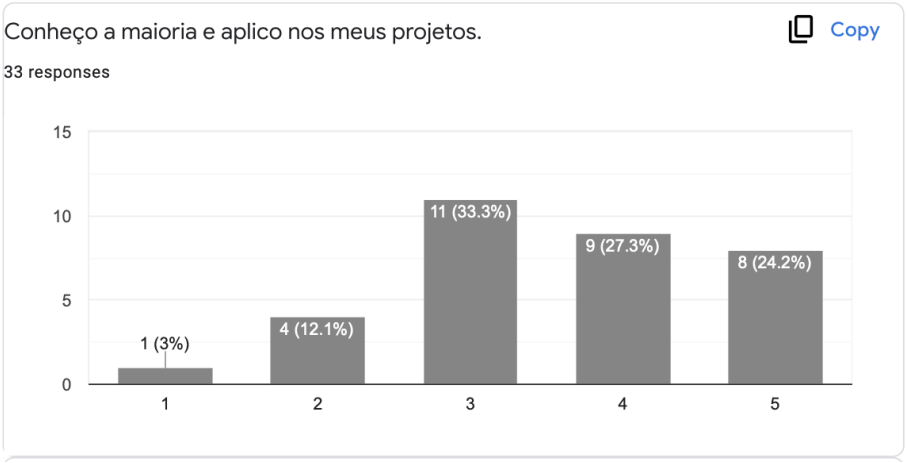

Figura 3.17: Pergunta: "Conheço a maioria e aplico nos meus projetos". Fonte: Google Forms.

quantidade de participantes que acredita que não é necessariamente uma questão de falta de tempo. Já a figura 3.22 ajuda a entender o quão relevante são, na visão dos participantes, cada um dos conceitos apresentados diante de seus respectivos projetos de software ao estabelecer uma graduação de importância contra a entrega de software de qualidade e que cumpre a estimativa de tempo estipulada.

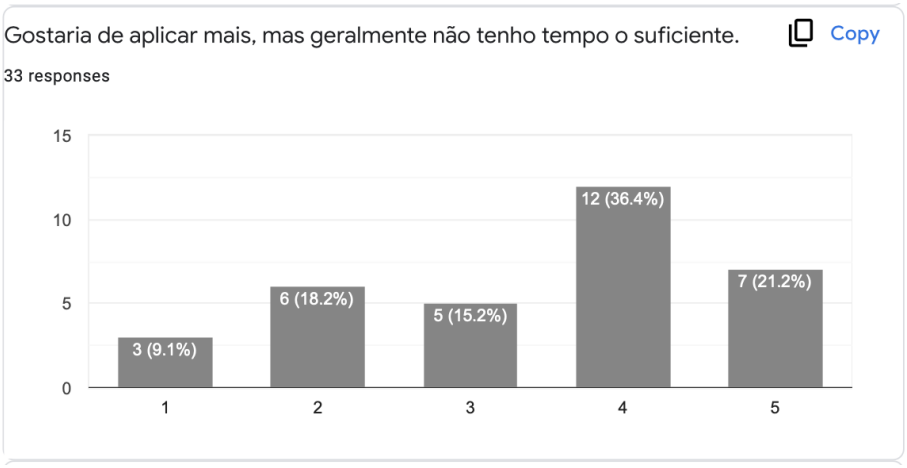

Figura 3.18: Pergunta: "Gostaria de aplicar mais, mas geralmente não tenho tempo o suficiente". Fonte: Google Forms.

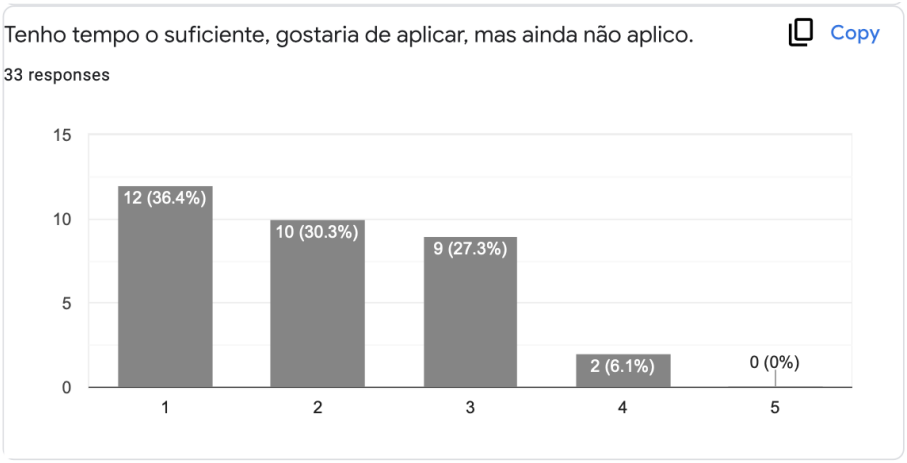

Figura 3.19: Pergunta: "Tenho tempo o suficiente, gostaria de aplicar, mas ainda não aplico". Fonte: Google Forms.

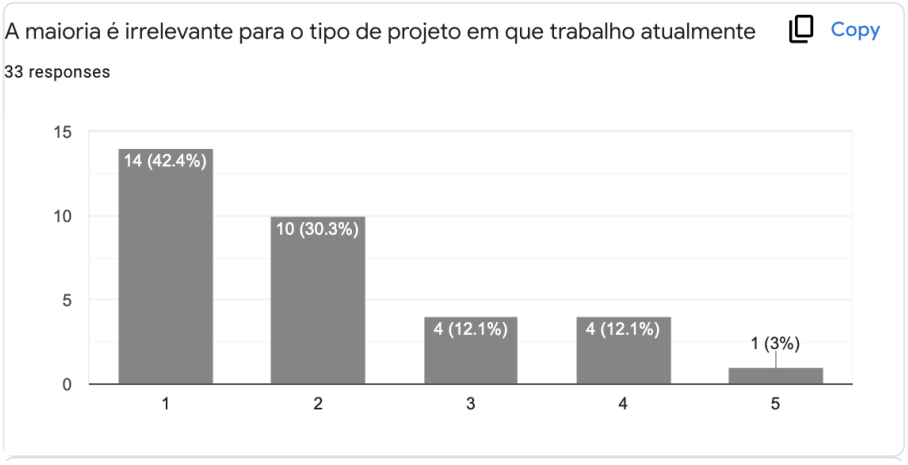

Figura 3.20: Pergunta: "A maioria é irrelevante para o tipo de projeto em que trabalho atualmente". Fonte: Google Forms.

O objetivo da questão apresentada na figura 3.24 é verificar se os participantes entendem que, nas empresas de tecnologia, a legibilidade e facilidade em estender e manter o código do projeto é frequentemente mais importante do que a quantidade de recurso computacional

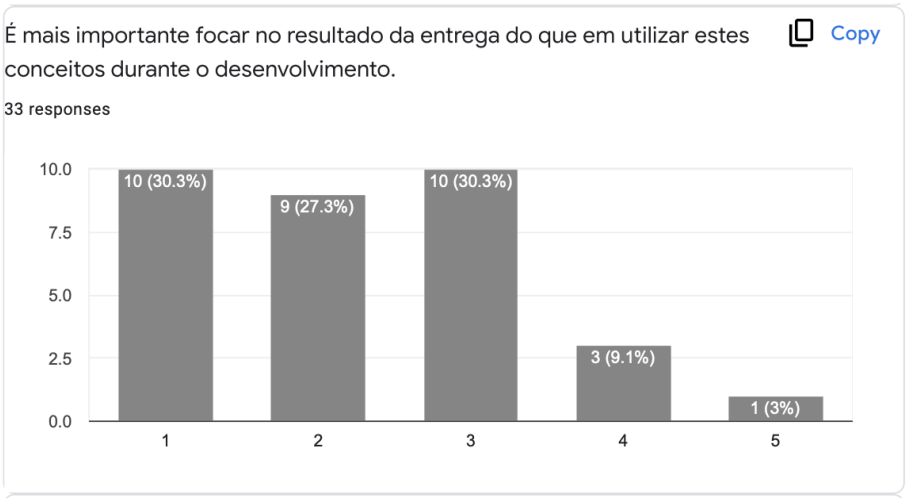

Figura 3.21: Pergunta: "É mais importante focar no resultado da entrega do que em utilizar estes conceitos durante o desenvolvimento". Fonte: Google Forms.

Figura 3.22: Pergunta: "É possível entregar software de qualidade e entregar rápido mesmo sem utilizar a maioria destes conceitos durante o desenvolvimento". Fonte: Google Forms.

utilizado – contanto que não prejudique a experiência do usuário. Isso é feito estabelecendo uma relação de preferência entre Complexidade Ciclomática e Complexidade Algorítmica, sendo a primeira intimamente relacionada à legibilidade, facilidade de manutenção, extensão e escrita de testes; enquanto a segunda está relacionada a um menor consumo de recurso computacional e melhora na experiência do usuário (no caso onde o software em questão tem contato direto ou indireto com os usuários de forma síncrona). Este objetivo está relacionado, ainda, aos resultados da questão apresentados na figura 3.25, que visa verificar se os participantes preferem projetos eficientes ou projetos que sejam de fácil extensão e manutenção.

O objetivo das perguntas ilustradas nas figuras 3.27 e 3.28 é contrastar suas respostas com a pergunta ilustrada na figura 3.36, confirmando ou não a hipótese de que os desenvolvedores sabem a importância de planejar, entretanto não planejam, corroborando com a tese da simples falta de prática contra a comum tese da falta de tempo (usada como justificativa pela maioria, de forma equivocada).

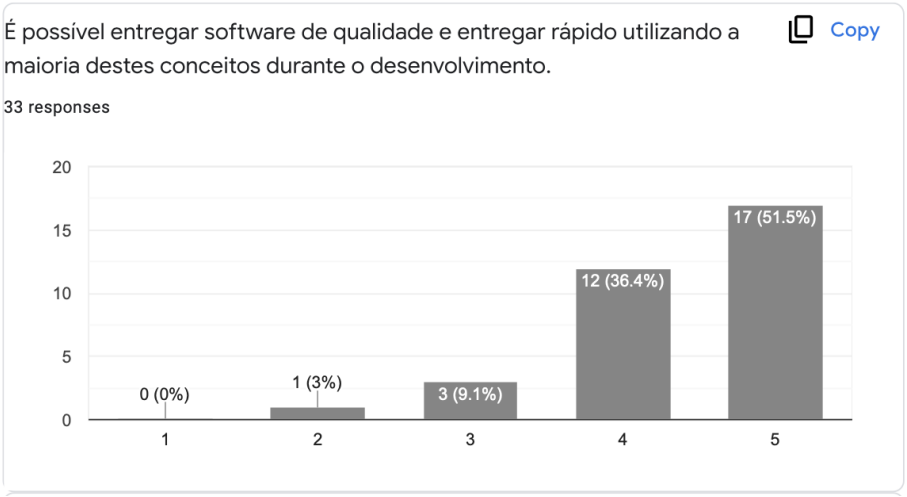

Figura 3.23: Pergunta: "É possível entregar software de qualidade e entregar rápido utilizando a maioria destes conceitos durante o desenvolvimento". Fonte: Google Forms.

Figura 3.24: Pergunta: "Qual dos conceitos abaixo você considera mais importante no código dos seus projetos atuais?". Fonte: Google Forms.

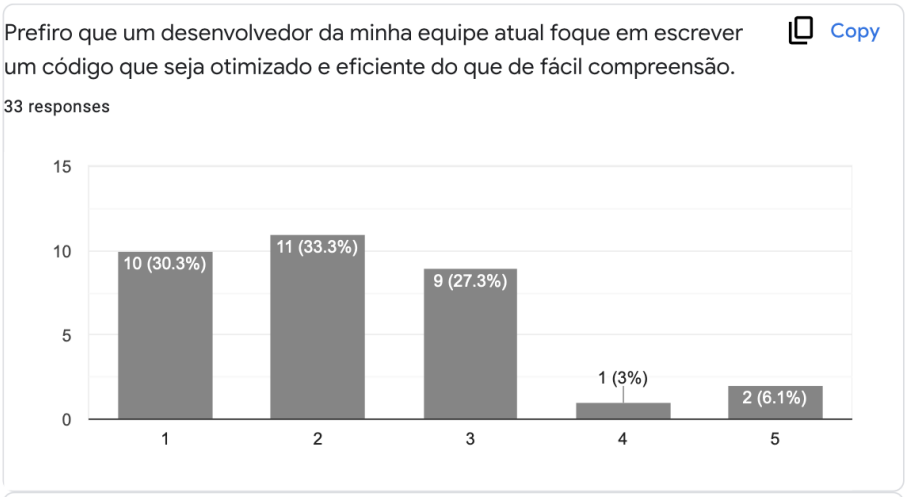

Figura 3.25: Pergunta: "Prefiro que um desenvolvedor da minha equipe atual foque em escrever um código que seja otimizado e eficiente do que de fácil compreensão". Fonte: Google Forms.

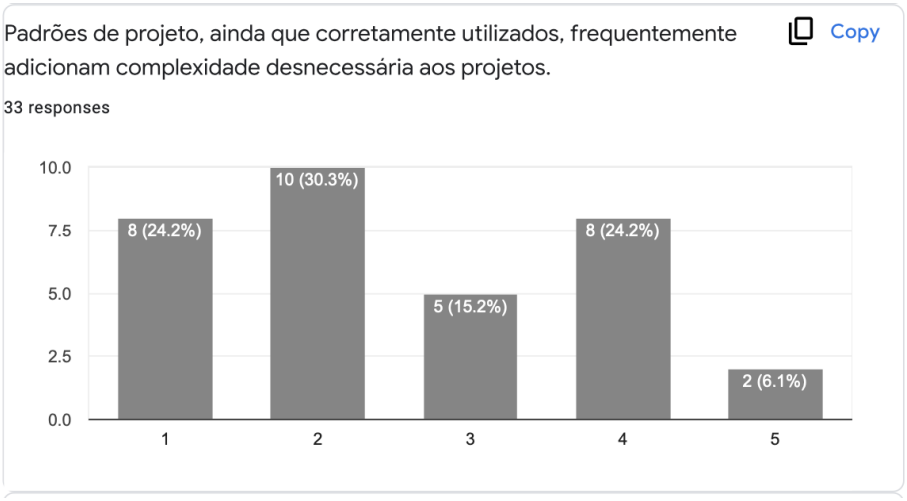

Figura 3.26: Pergunta: "Padrões de projeto, ainda que corretamente utilizados, frequentemente adicionam complexidade desnecessária aos projetos". Fonte: Google Forms.

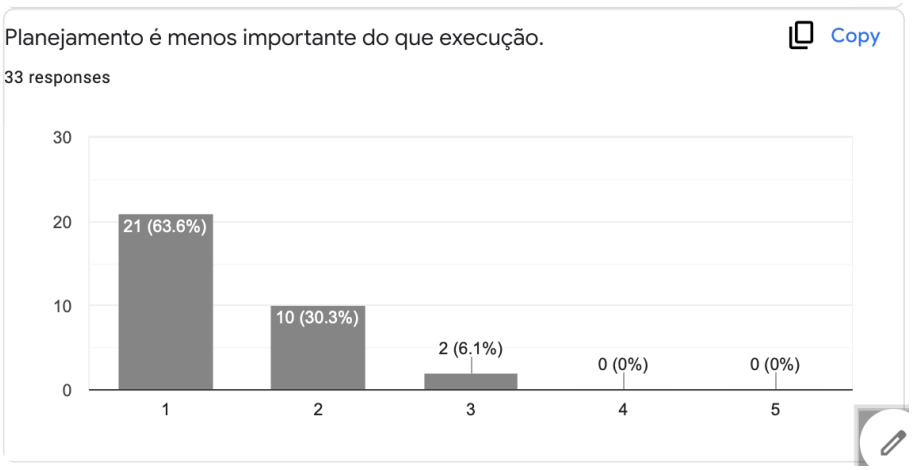

Figura 3.27: Pergunta: "Planejamento é menos importante do que execução". Fonte: Google Forms.

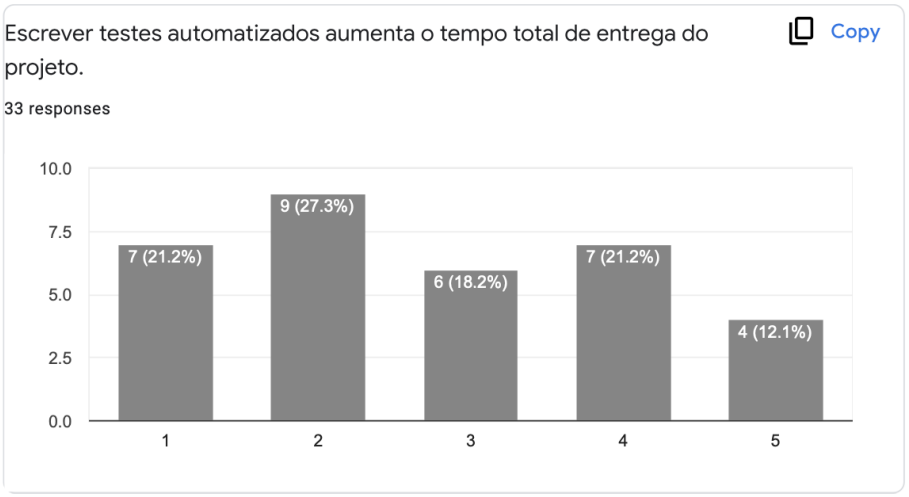

Figura 3.28: Pergunta: "Escrever testes automatizados aumenta o tempo total de entrega do projeto". Fonte: Google Forms.

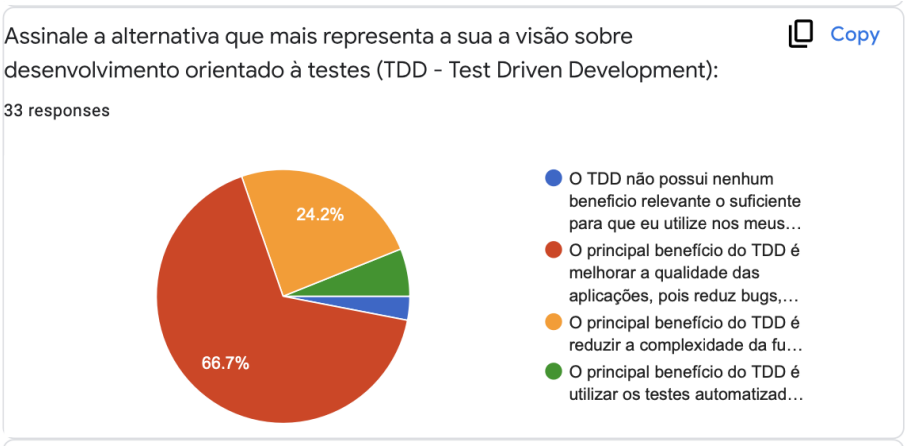

Figura 3.29: Pergunta: "Assinale a alternativa que mais representa a sua visão sobre desenvolvimento orientado à testes (TDD - *Test Driven Development*)". Fonte: Google Forms.

Já a pergunta ilustrada na figura 3.29 tem como principal objetivo entender a visão dos desenvolvedores quanto à real utilidade da prática de desenvolvimento orientado a testes – especificamente quanto à escrever casos de teste antes da funcionalidade que será testada. Esta pergunta surge diante da hipótese – posteriormente comprovada – de que os desenvolvedores, no geral, não entendem o real benefício de utilizar o TDD durante o desenvolvimento de um projeto de software, o que pode estar relacionado ao fato de que a grande maioria não aplica a técnica.

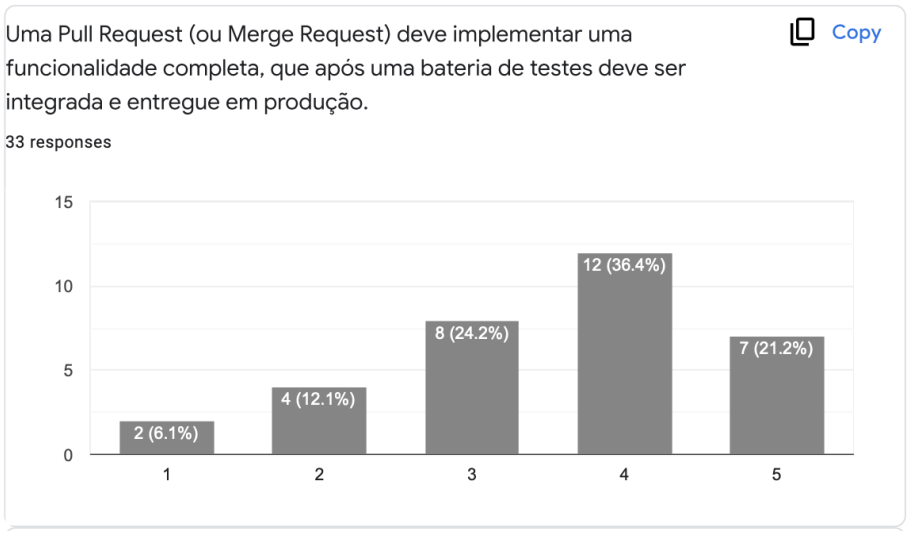

Figura 3.30: Pergunta: "Um *Pull Request* (ou *Merge Request*) deve implementar uma funcionalidade completa, que após uma bateria de testes deve ser integrada e entregue em produção.". Fonte: Google Forms.

As perguntas ilustradas nas figuras 3.30, 3.31 e 3.32 visam entender a visão dos participantes quanto à prática de CI/CD, contrastando com a técnica oposta, que seriam *releases* maiores de grandes porções de código entregues ao mesmo tempo.

Já as figuras 3.33, 3.34 e 3.35 ilustram as perguntas que tem como objetivo elucidar a visão dos participantes quanto aos motivos da dificuldade em entender, manter e estender código feito pro outros desenvolvedores. Isso é importante para saber se os participantes enxergam que as práticas aqui citadas são de fato importantes para melhorar a qualidade do software entregue.

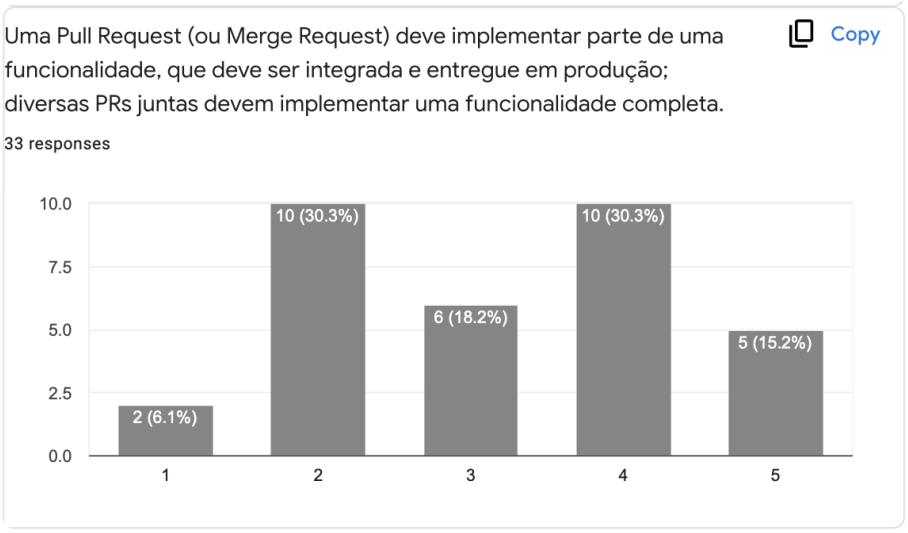

Figura 3.31: Pergunta: "Um *Pull Request* (ou *Merge Request*) deve implementar parte de uma funcionalidade, que deve ser integrada e entregue em produção; diversas PRs juntas devem implementar uma funcionalidade completa". Fonte: Google Forms.

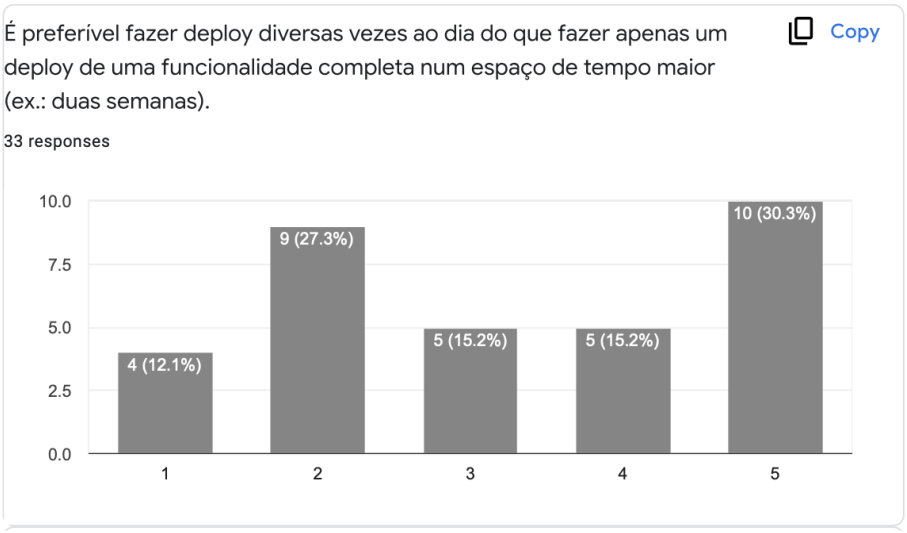

Figura 3.32: Pergunta: "É preferível fazer *deploy* diversas vezes ao dia do que fazer apenas um *deploy* de uma funcionalidade completa num espaço de tempo maior (ex.: duas semanas)". Fonte: Google Forms.

A pergunta cujos resultados são apresentados na figura 3.37 ajuda a verificar o nível familiaridade dos participantes quanto aos principais conceitos relacionados à arquitetura moderna de software. Para entender e aplicar estes conceitos em um projeto em produção, um desenvolvedor necessariamente precisa entender uma série de outros conceitos importantes aqui apresentados, especialmente os citados na sessão que discorre sobre SOLID e sobre Arquitetura Limpa.

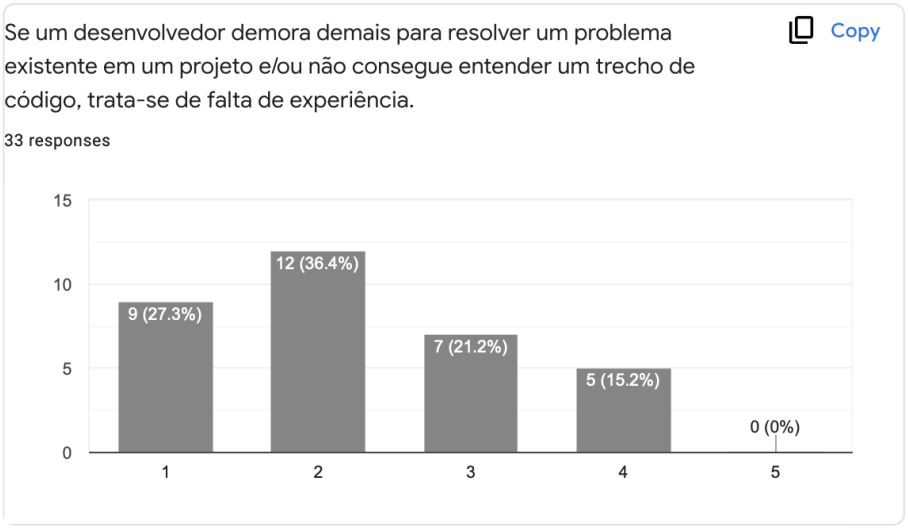

Figura 3.33: Pergunta: "Se um desenvolvedor demora demais pra resolver um problema existente em um projeto e/ou não consegue entender um trecho de código, trata-se de falta de experiência". Fonte: Google Forms.

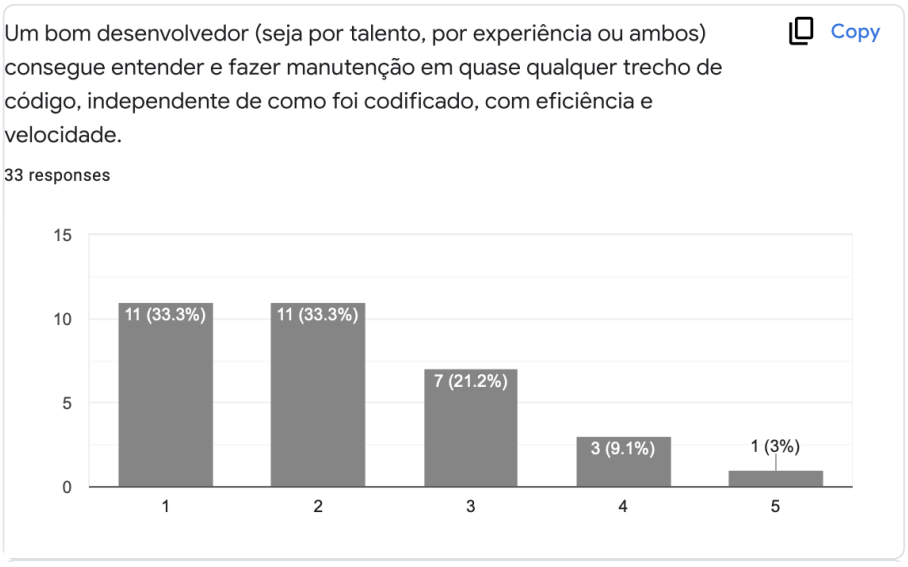

Figura 3.34: Pergunta: "Um bom desenvolvedor (seja por talento, experiência ou ambos) consegue entender e fazer manutenção em quase qualquer trecho de código, independente de como foi codificado, com eficiência e velocidade". Fonte: Google Forms.

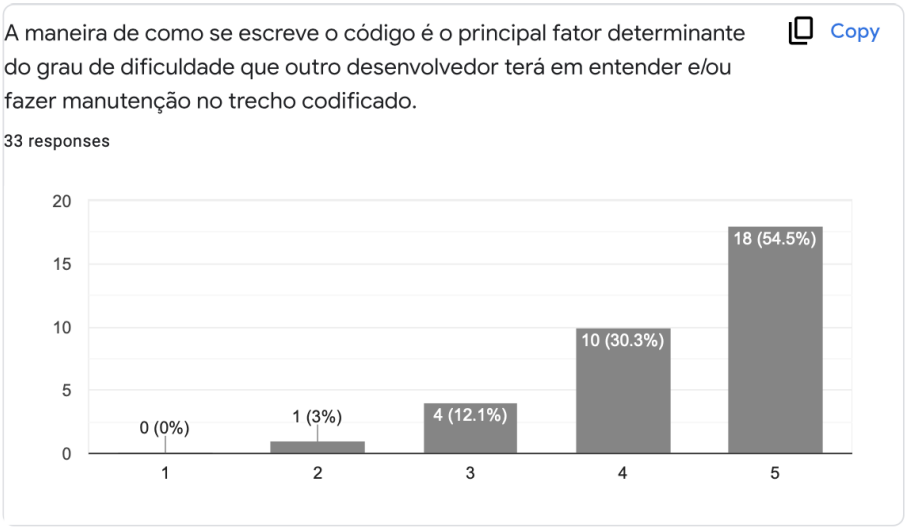

Figura 3.35: Pergunta: "A maneira de como se escreve o código é o principal fator determinante do grau de dificuldade que outro desenvolvedor terá em entender e/ou fazer manutenção no trecho codificado". Fonte: Google Forms.

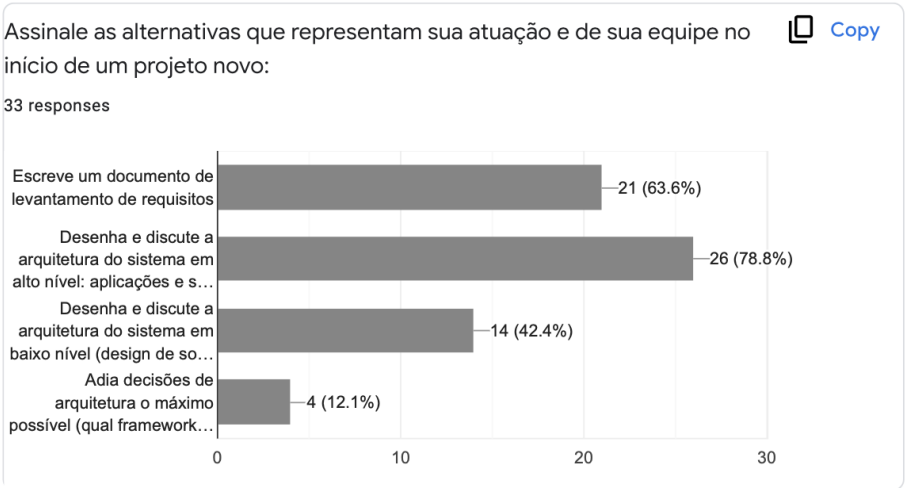

Figura 3.36: Pergunta: "Assinale as alternativas que representam sua atuação e de sua equipe no início de um projeto novo". Fonte: Google Forms.

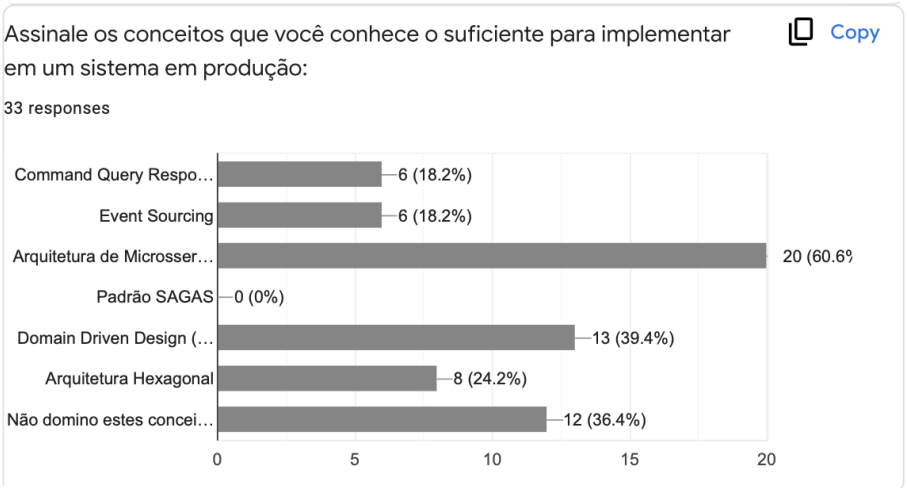

Figura 3.37: Pergunta: "Assinale os conceitos que você conhece o suficiente para implementar em um sistema em produção". Fonte: Google Forms.

4 RESULTADOS E DISCUSSÃO

Podemos observar que apesar da maioria dos participantes ser bastante jovem, já que a faixa de 18 a 35 anos representa 77,9%, o tempo de experiência fica entre 3 a 6 anos para 63,6% dos participantes, e desta forma podemos inferir que a linguagem, os termos técnicos e os conceitos básicos utilizados no questionário são de amplo conhecimento dos profissionais que responderam. É interessante observar, ainda, que as formações acadêmicas mais populares entre os participantes são Bacharelado em Ciência da Computação (42,4%), Tecnólogo em Análise e Desenvolvimento de Sistemas (24,4%) e Bacharelado em Sistemas de Informação (21,2%), com 54,5% participantes advindos de universidades públicas, e nenhum deles possuindo formação acadêmica em áreas não relacionadas à tecnologia da informação.

A linguagem de programação mais utilizada por um profissional da área possui certa relação com a qualidade do código dos projetos, já que existem linguagens de alto nível que são mais permissivas e possuem funcionalidades que facilitam a não observação de certas práticas de desenvolvimento que garantem a qualidade e a eficiência do projeto à longo prazo. Podemos, inclusive, observar que as linguagens mais permissivas e de alto nível na lista são as mais populares entre os participantes, como JavaScript (63,6%), PHP (51,5%) e Python (41,4%). Apenas 61,9% dos programadores JavaScript utilizam TypeScript, que é um superconjunto do JavaScript que reduz a permissibilidade da linguagem drasticamente à medida que possibilita o uso de tipagem estática. A lista completa de linguagens dadas como opção na questão foi retirada do *Developer Survey* 2021 do *Stack Overflow* (Overflow, 2021), que, entre outros temas, estabelece um ranking de popularidade das linguagens entre os oitenta mil entrevistados. Foi, entretanto, adicionada a possibilidade do participante inserir novas opções, caso suas principais linguagens não estivessem listadas. Apenas três novas linguagens foram inseridas pelos participantes: Apex, Dataflex e SAS.

Podemos observar que a maioria dos participantes conhece e utiliza as práticas listadas, com exceção do Desenvolvimento Orientado à Testes (TDD), da Metodologia *Twelve-Factor App* e da redução de Complexidade Ciclomática e Espacial. O TDD chama atenção pois 63,6% dos participantes afirmam conhecer, não utilizar, mas possuir vontade de utilizar. Este resultado pode estar relacionado com o fato de que 56,6% dos participantes afirmam que gostariam de aplicar as práticas listadas, mas não possuem tempo o suficiente. Como discutido anteriormente, é uma lenda que o uso de boas práticas de engenharia aumentam o tempo total de entrega de um software, e isso pode estar tão intrínseco no imaginário dos profissionais por conta da falta de prática na aplicação das técnicas, que acaba gerando um elevado tempo inicial de aprendizado que muitas vezes não pode ser pago por conta dos prazos apertados dos projetos em um ambiente competitivo e voltado ao resultado como o que se encontra a indústria de tecnologia hoje. Isso é especialmente explicitado pelo fato de que 87,9% dos participantes concordam que é possível

entregar software de qualidade e entregar rápido utilizando a maioria destes conceitos durante o desenvolvimento e pelo fato de que 48,5% acreditam que escrever testes automatizados não aumenta o tempo total de entrega do projeto (com 18,2% de abstenções quanto à esta questão em específico), portanto, na percepção coletiva, aparentemente não trata-se de falta de tempo propriamente dito, mas de falta de prática. A falta de prática, é, inclusive, o que provavelmente levou 66,7% dos participantes a afirmarem que "o principal benefício do TDD é melhorar a qualidade das aplicação, pois reduz *bugs*, garante que o código funciona como esperado e impede que modificações incorretas vão para o ambiente de produção", afinal, um desenvolvedor que conhece e utiliza TDD no dia a dia provavelmente entende que, na verdade, o principal benefício da prática está mais ligado ao entendimento da funcionalidade que está sendo implementada, facilitando o processo de desenvolvimento à medida que ajuda a organizar melhor as ideias e à implementar um código desacoplado por padrão.

No caso da Metodologia *Twelve-Factor App* e da Redução de Complexidade Ciclomática e Espacial, o resultado é diferente: a maioria dos entrevistados não conhece os termos. Tanto a Complexidade Ciclomática quanto Espacial são comumente abordadas em âmbito acadêmico durante os anos de formação, o que indica que talvez seja um problema de qualidade curricular ou de formação acadêmica. Abordar os termos durante os anos de faculdade certamente faria com que parte destes profissionais aplicasse ambas em seus projetos, ou no mínimo tivessem conhecimento quanto às vantagens. Já a Metodologia *Twelve-Factor App* é específica para profissionais que trabalham com *software as a service* (SAAS), o que pode justificar o fato de que grande parte dos entrevistados nunca ouviu falar do termo, já que existem diversas outras áreas dentro da indústria de tecnologia da informação.

Como esperado, já que a maioria não conhece o temo "Complexidade Ciclomática", a questão que tem como objetivo verificar uma preferência entre priorizar Complexidade Ciclomática ou priorizar Complexidade Algorítmica teve 39,4% (maioria) de respostas "não tenho certeza". Entre os que conhecem ambos os termos, a maioria (33,3%) prefere uma baixa Complexidade Algoritimica à uma baixa Complexidade Ciclomática (27,3%) em seus projetos atuais. Isso é um mal sinal, visto a prioridade da maioria das empresas de tecnologia está relacionada à legibilidade e à facilidade de extensão e manutenção, mais do que o uso de recursos computacionais, como anteriormente discutido. Este resultado é diferente, entretanto, na pergunta sobre código otimizado e eficiente contra código de fácil compreensão: 63,6% acreditam que um código de fácil compreensão é mais importante do que um código eficiente, e este resultado pode ter sido obtido por conta dos 39,4% que não conhecem o termo Complexidade Ciclomática da pergunta anterior, e portanto abstiveram-se. Isto é no entanto preocupante, já que escrever um código de fácil compreensão está intrinsecamente ligado à redução de Complexidade Ciclomática, e pode indicar que apesar de existir uma vontade de escrever códigos menos complexos, aparentemente não há ação quanto à isso, ou há uma baixa prioridade quando comparado com a entrega do projeto em si.

É interessante ressaltar que a maioria dos participantes (84,8%) concorda que a maneira de como se escreve o código é o principal fator determinante do grau de dificuldade que outro desenvolvedor terá em entender e/ou fazer manutenção no trecho codificado, e que nem sempre um desenvolvedor talentoso e/ou experiente consegue fazer manutenção em um trecho com eficiência e velocidade (66,6%). Isso confirma que, apesar de não estarem familiarizados com todas as práticas que podem facilitar a manutenibilidade de um projeto de software, os participantes tendem a achar suas aplicações desejáveis, novamente voltado à suspeita de que trata-se de uma questão de falta de prática. Ainda quanto à isso, 93,9% dos participantes concordam que planejamento é mais importante do que execução, entretanto, apenas 42,4% afirmam desenhar e discutir a arquitetura antes do início do desenvolvimento, por exemplo.

5 CONCLUSÃO

Os resultados da pesquisa esclarecem o fato de que as boas práticas listadas frequentemente podem não ser observadas pelas equipes de engenharia nas empresas de tecnologia do Brasil. Podemos, ainda, concluir que a principal causa aparenta ser a falta prática, a simples não aplicação dos conceitos listados, visto que a maioria possui conhecimento dos mesmos. Entretanto, este fato parece não ser reconhecido pelo profissionais, que acreditam tratar-se de uma questão de falta de tempo — hipótese esta que é invalidada pelos próprios participantes ao afirmarem que acreditam ser possível entregar software rápido e com qualidade mesmo observado cada uma das boas práticas listadas, e pela própria experiência dos que aplicam diariamente, bem como os resultados de trabalhos correlatos.

A solução para este problema é vencer a inércia e fazer o esforço inicial necessário para aprender e aplicar as práticas no dia a dia. Entretanto, para além de uma atitude individual de cada engenheiro, as empresas de tecnologia podem trabalhar maneiras de incentivar a aplicação dos conceitos de diversas formas. Uma das maneiras de fazer isso é a adoção incremental das práticas através de regras simples. Uma equipe pode, por exemplo, instituir uma regra que estabelece que cada desenvolvedor escreva ao menos um teste automatizado toda vez que for adicionar um trecho novo de código à base. Uma regra destas não engessa o desenvolvimento, é fácil de compreender, e pode ser imediatamente aplicada à qualquer projeto. Com o tempo, o número de testes pode naturalmente aumentar, já que os colaboradores estarão familiarizados com o processo de escrita de testes automatizados. A mesma técnica pode ser reaproveitada para todas as boas práticas listadas neste trabalho, constituindo uma ótima maneira de mudar esta realidade.

A observação destas práticas tem o real potencial de ajudar a criar uma abordagem mais sistemática para as disciplinas relacionadas a engenharia de software, ajudando a planejar, medir, pensar à longo prazo, obter sucesso nos projetos e principalmente economizar tempo, e, portanto recursos financeiros. Com isso, a engenharia de software pode dar saltos cada vez maiores em direção à um estado de maturidade em que outras engenharias já se encontram hoje em dia (como a mecânica e a civil), gerando cada vez mais valor para a sociedade à medida que cria produtos e serviços valiosos gerando emprego, renda, movimentando a economia, e, portanto contribuindo para o bem estar social.

REFERÊNCIAS

de Português, D. D. O. (2021). Complexo. https://www.dicio.com.br/complexo/. Acessado em 06/05/2022.

Dijkstra, E. W. (1972). The humble programmer. *Commun. ACM*, 15(10):859–866.

Fowler, M. (2013). Telldontask. https://martinfowler.com/bliki/TellDontAsk.html. Acessado em 06/05/2022.

Freeman, S. (2009). *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley Professional.

Jacobson, I. (1992). *Object Oriented Software Engineering: A Use-Case Driven Approach*. McGraw-Hill Science/Engineering/Math.

Jones, C. (2004). Software project management practices: Failureversus success.

Martin, R. C. (2002). *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall.

Martin, R. C. (2008). *Clean Code, A Handbook of Agile Software Craftsmanship*. Pearson Education.

Martin, R. C. (2011). Screaming architecture. https://blog.cleancoder.com/uncle-bob/2011/09/30/Screaming-Architecture.html. Acessado em 06/05/2022.

Martin, R. C. (2019). *Arquitetura limpa: O guia do artesão para estrutura e design de software*. Alta Books.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320.

Overflow, S. (2021). Developer survey 2021. https://insights.stackoverflow.com/survey/2021. Acessado em 06/05/2022.

Palermo, J. (2008). The onion architecture: part 1. https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/. Acessado em 06/05/2022.

Wiggins, A. (2017). The twelve-factor app. https://12factor.net. Acessado em 06/05/2022.